Documents for the PWB/UNIX Time-Sharing System

Edition 1.0

T. A. Dolotta
R. C. Haight
E. M. Piskorik

Editors

October 1977

Bell Telephone Laboratories, Incorporated

The enclosed documents were set on a Graphic Systems, Inc. phototypesetter driven by the TROFF formatting program. Their text was prepared using the ED text editor.

Documents for the PWB/UNIX Time-Sharing System

Annotated Table of Contents

Each item carries the date of its latest revision; most items also give the number of their last page.

G. General:

G.1 *PWB/UNIX—Overview and Synopsis of Facilities* (6/77)
T. A. Dolotta and R. C. Haight (p. 18)
Summarizes the salient features of Programmer's Workbench/UNIX, a program development and text processing facility.

G.2 *The UNIX Time-Sharing System* (7/74)
D. M. Ritchie and K. Thompson (p. 16)
Reprinted from *Comm. ACM*. Good overview of UNIX, but written long ago.

G.3 *The UNIX Time-sharing System—A Retrospective* (1/77)
D. M. Ritchie (p. 14)
A more recent discussion.

G.4 *PWB/UNIX Papers from the Second Intern. Conf. on Software Engineering* (10/76)
T. A. Dolotta *et al.* (p. 25)
Reprinted from that conference's proceedings. Four papers that describe various aspects of PWB/UNIX.

B. Basic User Information:

B.0 *PWB/UNIX User's Manual—Edition 1.0* (5/77)
T. A. Dolotta, R. C. Haight, and E. M. Piskorik, eds.
Describes all commands, subroutines, and system calls. Furnished as a separate volume. Available on-line.

B.1 *PWB/UNIX Documentation Roadmap* (5/77)
J. R. Mashey (p. 7)
A terse, up-to-date outline of useful documents and information sources. Available on-line.

B.2 *PWB/UNIX Beginner's Course* (12/77)
M. E. Pearlman and S. H. Strauss
An outline with view graphs.

B.3 *A Tutorial Introduction to the UNIX Text Editor* (10/74)
B. W. Kernighan (p. 10)
Expectedly elementary, but very useful for beginners.

B.4 *Advanced Editing on UNIX* (8/76)
B. W. Kernighan (p. 16)
Meant to help secretaries, typists, and programmers make effective use of UNIX facilities for preparing and editing text.

B.5 *PWB/UNIX Shell Tutorial* (9/77)
J. R. Mashey (p. 25)
Describes the PWB/UNIX command interpreter.

B.6 *UNIX for Beginners* (10/74)
B. W. Kernighan (p. 14)
A slightly dated tutorial.

B.7 *UNIX Programming* (10/75)
B. W. Kernighan and D. M. Ritchie (p. 17)
Introduction to programming on UNIX. The emphasis is on how to write programs that interface with the operating system. Does *not* cover material in *A New Input/Output Package* (item B.10 below).

B.8 C *Reference Manual* (5/77)
D. M. Ritchie (p. 32)
Terse, but complete.

B.9 *Programming in* C*—A Tutorial* (5/75)
B. W. Kernighan (p. 27)
Should be read before tackling the C *Reference Manual* (item B.8 above).

B.10 *A New Input-Output Package* (7/77)
D. M. Ritchie (p. 6)
Should be used for all new C programs.

B.11 *A General-Purpose Subroutine Library for PWB/UNIX* (7/77)
A. L. Glasser (p. 7)
Complements *A New Input/Output Package* (item B.10 above).

B.12 *Guide to IBM Remote Job Entry for PWB/UNIX Users* (9/77)
A. L. Sabsevitz (p. 7)
Describes the RJE facility between a PWB/UNIX system and an IBM System/370.

B.13 *SCCS/PWB User's Manual* (11/77)
L. E. Bonanni and A. L. Glasser (p. 22)
Describes the Programmer's Workbench Source Code Control System.

T. Text Processing, Formatting, and Typesetting:

T.1 *NROFF/TROFF User's Manual* (5/77)
J. F. Ossanna (p. 34)
NROFF and TROFF are text processors. NROFF formats text for a variety of typewriter-like terminals. TROFF formats text for a Graphic Systems, Inc. phototypesetter.

T.2 *PWB/MM—Programmer's Workbench Memorandum Macros* (10/77)
D. W. Smith and J. R. Mashey (p. 56)
User's guide and reference manual for PWB/MM, a general-purpose package of text formatting macros for use with NROFF and TROFF.

T.3 *Typing Documents with PWB/MM* (10/77)
D. W. Smith and E. M. Piskorik (p. 16)
A fanfold card that fits into a pocket(book).

T.4 *PWB/MM Tutorial* (12/77)
N. W. Smith
Introduction to PWB/UNIX text processing.

T.5 *Tbl—A Program to Format Tables* (9/77)
M. E. Lesk (p. 17)
Preprocessor for TROFF or NROFF that makes even very complex tables easy to specify.

T.6 *A TROFF Tutorial* (8/76)
B. W. Kernighan (p. 13)
Introduction to the most basic use of TROFF (and, by implication, NROFF).

T.7 *Typesetting Mathematics—User's Guide (Second Edition)* (6/76)
B. W. Kernighan and L. L. Cherry (p. 11)
Describes the EQN and NEQN preprocessors for TROFF and NROFF, respectively. They allow one to typeset complex formulae, equations, arrays, etc., both in-line and displayed.

T.8 *New Graphic Symbols for EQN and NEQN* (9/76)
C. Scrocca (p. 8)
Defines a set of special characters frequently used in technical documents. Shows how to use them and discusses what is involved in making a special character in NROFF and TROFF.

T.9 *PWB/UNIX View Graph and Slide Macros* (12/77)
T. A. Dolotta and D. W. Smith
Greatly eases the task of making transparencies with TROFF.

A. Additional Facilities:

A.1 *BC—An Arbitrary Precision Desk Calculator Language* (5/75)
L. L. Cherry and R. Morris (p. 14)
A language and a compiler for doing arbitrary-precision arithmetic.

A.2 *DC—An Interactive Desk Calculator* (5/75)
R. Morris and L. L. Cherry (p. 8)
Interactive desk calculator program that does arbitrary-precision integer arithmetic.

A.3 *YACC—Yet Another Compiler Compiler* (5/75)
S. C. Johnson (p. 30)
Generates parsers from context-free language specifications.

A.4 *LEX—Lexical Analyzer Generator* (4/77)
M. E. Lesk and E. Schmidt (p. 13)
LEX helps write programs whose control flow is directed by instances of regular expressions in the input stream.

A.5 *RATFOR—A Preprocessor for a Rational Fortran* (1/77)
B. W. Kernighan (p. 12)
IF-ELSE, WHILE, and other useful control structures.

A.6 *The M4 Macro Processor* (4/77)
B. W. Kernighan and D. M. Ritchie (p. 6)
A general-purpose macro language; can be used as a preprocessor for RATFOR, C, etc.

A.7 *Make—A Program for Maintaining Computer Programs* (4/77)
S. I. Feldman (p. 9)
Make provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files.

I. Internals, Operations, and Administration:

I.1 *Setting Up PWB/UNIX* (9/77)
R. C. Haight, W. D. Roome, and L. A. Wehr (p. 16)
Procedures used to install PWB/UNIX on the PDP-11/45 or /70 and the steps necessary to regenerate all of the PWB/UNIX programs.

I.2 *Administrative Advice for PWB/UNIX* (10/77)
R. C. Haight (p. 8)
Hints for approaching operational serenity.

I.3 *PWB/UNIX Operations Manual* (9/77)
M. E. Pearlman (p. 36)
Describes the daily routine at the console. Text (but not pictures) available on-line.

I.4 *Repairing Damaged PWB/UNIX File Systems* (11/77)
P. D. Wandzilak
Comes in handy after a power failure, etc.

I.5 *PWB/UNIX RJE Administrator's Guide* (12/77)
A. L. Sabsevitz
What to do when it breaks.

I.6 *The UNIX I/O System* (6/74)
D. M. Ritchie (p. 9)
Describes how to write device drivers for UNIX.

I.7 *On the Security of UNIX* (6/74)
D. M. Ritchie (p. 4)
A short, but enlightening, discussion.

I.8 *UNIX Assembler Reference Manual* (6/73)
D. M. Ritchie (p. 12)
As a last resort ...

I.9 *PWB/UNIX Manual Page Macros* (8/77)
E. M. Piskorik (p. 7)
Tells how to make PWB/UNIX User's Manual pages.

R. Recommended Reading—not Included:

R.1 *Software Tools*
B. W. Kernighan and P. J. Plauger (p. 338)
Addison-Wesley, Reading, MA; 1976.

R.2 The UNIX Command Language
K. Thompson
In *Structured Programming—Infotech State of the Art Report.* Infotech International Limited, Nicholson House, Maidenhead, Berkshire, England; 1976; pp. 375-84.

PWB/UNIX Documentation Roadmap

J. R. Mashey

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

A great deal of documentation exists for PWB/UNIX. It has different formats, is contributed by many different people, and is modified frequently. New users are often overcome by the volume and distributed nature of the documentation. This "roadmap" attempts to be a terse, up-to-date outline of crucial documents and information sources.

Numerous people have contributed comments and information for this "roadmap," in order to make it as helpful as possible for PWB/UNIX users. *However, many of these comments are accurate only with regard to PWB/UNIX and may well be totally inapplicable to other versions of UNIX.*

1.1 Things to Do

See a local PWB/UNIX system administrator to obtain a "login name" and get other appropriate system information.

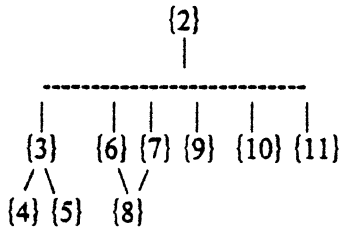
1.2 Notation Used in this Roadmap

{*N*} — Section N in this "roadmap."
++ — item required for everyone.
\+ — item recommended for most users.

All other items are optional and depend on specific interest (a list of relevant documents appears in the Table of Contents of *Documents for the PWB/UNIX Time-Sharing System*).

Items in Section *N* of the *PWB/UNIX User's Manual* are referred to by *name*(N).

1.3 Prerequisite Structure of Following Sections



2. BASIC INFORMATION

Don't do anything else until you have learned most of this section. You must know how to log onto the system, make your terminal work correctly, enter and edit files, and perform basic operations on directories and files.

2.1 PWB/UNIX User's Manual ++

- Read *Introduction* and *How to Get Started.*
- Look through Section I to become familiar with command names.
- Note the existence of the *Table of Contents* and of the *Permuted Index.*

Section I will be especially needed for reference use.

2.2 UNIX for Beginners ++

2.3 A Tutorial Introduction to the UNIX Text Editor ++

2.4 Advanced Editing on UNIX +

2.5 PWB Papers from the Second International Conference on Software Engineering +

Gives an overview of the Programmer's Workbench.

2.6 Things to Do

- Do all the exercises found in {2.2} and {2.3}, and maybe {2.4}.
- Create a file named ".mail" in your login directory,* so that other people (such as system administrators) can send you mail. This can be done by:

  ```
  cp /dev/null .mail
  ```

- If you want some sequence of commands to be executed each time you log in, create a file named ".profile" in your login directory containing the commands you want executed. For more information, see *Initialization* in *sh*(I).
- Files in directory "/usr/news" contain recent information on various topics. To see what has been updated recently, type:

  ```
  ls -lt /usr/news
  ```

  and then print any files that look interesting. Other useful actions include:

mail -f /usr/news/.mail	gives recent history from primary system mailbox.
cat /usr/news/helpers	gives contacts and telephone numbers for counseling, file restorals, trouble reporting, and other services.
nroff -mm /usr/news/roadmap	prints current copy of this "roadmap."
cat /usr/news/terminals	gives recommendations on selection of computer terminals.

2.7 Manual Pages to Be Studied

The following commands are described in Section I of the *PWB/UNIX User's Manual,* and are used for creating, editing, moving (i.e., renaming), and removing files:

cat(I)	concatenate and print files (no pagination).
chdir(I)	change working (current) directory; a.k.a. *cd*(I).
cp(I)	make a copy of an existing file.
ed(I)	text editor.
ls(I)	list a directory; file names beginning with "." are not listed unless the "-a" flag is used.
mkdir(I)	make a (new) directory.
mv(I)	move (rename) file.
pr(I)	print files (paginated listings).
rm(I)	remove (delete) file(s).
rmdir(I)	remove directory(ies).

The following help you communicate with other users, make proper use of different kinds of terminals, and print manual pages on-line:

login(I)	sign on.
mail(I)	send mail to other users; inspect mail from them, or contents of the system mailbox.
man(I)	print pages of *PWB/UNIX User's Manual.*
stty(I)	set terminal options; i.e., inform the system about the hardware characteristics of your terminal.
tabs(I)	set tab stops on your terminal.
terminals(VII)	gives descriptions of commonly-used terminals.
who(I)	print list of users currently logged in.
write(I)	communicate with another (logged in) user.

* The directory you are in right after logging into the system.

3. BASIC TEXT PROCESSING AND DOCUMENT PREPARATION

You should read this section if you want to *use* existing text processing tools to write letters, memoranda, manuals, etc.

3.1 PWB/MM—Programmer's Workbench Memorandum Macros ++

This is a reference manual, and can be moderately heavy going for a beginner. Try out some of the examples, and stick close to the default options.

3.2 Typing Documents with PWB/MM ++

A handy fold-out.

3.3 NROFF/TROFF User's Manual +

Describes the text formatting language in great detail; look at the REQUEST SUMMARY, but don't try to digest the whole manual on first reading.

3.4 Documentation Tools and Techniques +

This overview of UNIX text processing methods is one of the papers from the Second International Conference on Software Engineering. (See {2.5} above).

3.5 Manual Pages to Be Studied

mm(I)	makes it easy to specify standard options to *nroff*(I).
nroff(I)	read to see formatter option flags.
spell(I)	identifies possible spelling errors.
tmac.name(VII)	list of text-formatting macro packages.
typo(I)	identifies possible typographical errors.

To obtain some special functions (e.g., reverse paper motion, subscripts, superscripts), you must either indicate the terminal type to *nroff* or post-process *nroff* output through one of the following:

450(I)	newer Diablo printer terminals, such as the DASI450, DIABLO 1620, XEROX 1700, etc.
col(I)	terminals lacking physical reverse motion, such as the Texas Instrument 700 series.
gsi(I)	older Diablo printer terminals, such as the GSI300, DASI300, DTC300, etc.
hp(I)	Hewlett-Packard 2640 terminals (HP2640A, HP2640B, HP2644A, HP2645A, etc.).

4. SPECIALIZED TEXT PROCESSING

The tools listed in this section are of a more specialized nature than those in {3}.

4.1 TBL—A Program to Format Tables +

Great help in formatting tabular data (see *tbl*(I)).

4.2 Typesetting Mathematics—User's Guide (2nd Edition) +

Read this if you need to produce mathematical equations. It describes the use of the equation setting commands *eqn*(I) and *neqn*(I).

4.3 A TROFF Tutorial

An introduction to formatting text with the phototypesetter.

4.4 Manual Pages to Be Studied

diffmark(I)	marks changes between versions of a file, using output of *diff*(I) to produce "revision bars" in the right margin.
eqn(I)	preprocessor for mathematical equations (phototypesetter).
neqn(I)	preprocessor for mathematical equations (terminals).
tbl(I)	preprocessor for tabular data.
troff(I)	formatter for phototypesetter.

5. ADVANCED TEXT PROCESSING

You should read this section if you need to *design* your own package of formatting macros or perform other actions beyond the capabilities of existing tools; {3} is a prerequisite, and familiarity with {4} is very helpful, as is an experienced advisor. It takes a great deal of effort to write a good package of macros for general use. Don't reinvent what you can borrow from an existing package (such as PWB/MM).

5.1 NROFF/TROFF User's Manual ++

Look at this in detail and try modifying the examples. If you are going to use the phototypesetter, do the same for *A TROFF Tutorial* ({4.3} above).

5.2 Things to Do

It is fairly easy to use the text formatters for simple purposes. A typical application is that of writing simple macros that print standard headings in order to eliminate repetitive keying of such headings. It is extremely difficult to set up general-purpose macro packages for use by large numbers of people. If possible, try to use an existing package or modify one as needed. Look at existing packages first—see *tmac.name*(VII).

5.3 Manual Pages to Be Studied

All pages mentioned in {3} and {4}.

6. COMMAND LANGUAGE (SHELL) PROGRAMMING

The Shell provides a powerful programming language for combining existing commands. This section should be especially useful to those who want to automate manual procedures and build data bases.

6.1 The UNIX Time-Sharing System ++

6.2 PWB/UNIX Shell Tutorial ++

6.3 Things to Do

If you want to create your own library of commands, create a ".path" file in your login directory, as described in *sh*(I).

6.4 Manual Pages to Be Studied

Read *sh*(I) first; the following pages give further details on commands that are most frequently used within command language programs:

echo(I)	echo arguments (typically to terminal).
equals(I)	Shell assignment command (for variables).
exit(I)	terminate command file.
expr(I)	evaluate an algebraic expression.
fd2(I)	redirect diagnostic output.
if(I)	conditional command.
next(I)	read command input from named file.
nohup(I)	run a command immune to communications line hang-up.
onintr(I)	handle interrupts in Shell files.
pump(I)	Shell data transfer command.
sh(I)	Shell (command interpreter).
shift(I)	adjust Shell arguments.
switch(I)	Shell multi-way branch command.
while(I)	Shell iteration command.

7. FILE MANIPULATION

In addition to the basic commands of {2}, many UNIX commands exist to perform various kinds of file manipulation. Small data bases can often be managed quite simply, by combining text processing (from {5}), command language programming {6}, and commands listed below in {7.2}.

7.1 Things to Do

This "roadmap" notes only the most frequently used commands. It is wise to scan Section I of the *PWB/UNIX User's Manual* periodically—you will often discover new uses for commands.

7.2 Manual Pages to Be Studied

The following are used to search or edit files in a single pass:

grep(I)	search a file for a pattern; more powerful and specialized versions include *egrep*(I), *fgrep*(I), and *rgrep*(I).
sed(I)	stream editor.
tr(I)	transliterate (substitute or delete specified characters).

The following compare files in different ways:

cmp(I)	compare files (byte by byte).
comm(I)	print lines common to two files, or lines that appear in only one of the two files.
diff(I)	differential file comparator (minimal editing for conversion).

The following combine files and/or split them apart:

ar(I)	archiver and library maintainer.
cpio(I)	general file copying and archiving.
csplit(I)	split file by context.
split(I)	split file into chunks of specified size.

These commands interrogate files and print information about them:

file(I)	determine file type (best guess).
od(I)	octal dump (and other kinds also).
wc(I)	word (and line) count.

Miscellaneous commands:

find(I)	search directory structure for specified kinds of files.
gath(I)	gather real and virtual files; alias for *send*(I).
help(I)	ask for help about a specific error message.
reform(I)	reformat "tabbed" files (often used to truncate lines).
sort(I)	sort or merge files.
tee(I)	copy single input to several output files.
uniq(I)	report repeated lines in a file, or obtain unique ones.

8. C PROGRAMMING

Try to use existing tools first, before writing C programs at all.

8.1 Programming in C—A Tutorial ++

Read; try the examples.

8.2 C Reference Manual ++

Terse but complete reference manual.

8.3 A New Input-Output Package +

Describes a new I/O package that is superseding many of the existing routines; write any new code using this package.

8.4 UNIX Programming +

8.5 YACC—Yet Another Compiler Compiler

8.6 LEX—Lexical Analyzer Generator

8.7 Make—A Program for Maintaining Computer Programs

8.8 Things to Do

The best way to learn C is to look at the source code of existing programs, especially ones whose functions are well known to you. Much code can be found in directory "/sys/source". In particular, directories "s1" and "s2" contain the source for most of the commands. Also, investigate directory "/usr/include".

8.9 Manual Pages to Be Studied

adb(I)	C debugger; more powerful (but more complex) than the older *cdb*(I).
cc(I)	C compiler.
cdb(I)	C debugger (for post-mortem core dumps and other debugging).
ld(I)	loader (you must know about some of its flags).
lex(I)	generate lexical analyzers.
make(I)	automate program regeneration procedures.
nm(I)	print name (i.e., symbol) list.
prof(I)	display profile data (used for program optimization).
regcmp(I)	compile regular expression.
strip(I)	remove symbols and relocation bits from executable file.
time(I)	time a command.
yacc(I)	parser generator.

9. IBM REMOTE JOB ENTRY (RJE)

This section is for those who use PWB/UNIX to submit jobs to remote computers.

9.1 Guide to IBM Remote Job Entry for PWB/UNIX Users +

9.2 Manual Pages to Be Studied

bfs(I)	big file scanner (scans RJE output).
csplit(I)	split file by context (often used to split RJE output).
fspec(V)	format specification in text files.
reform(I)	reformat files (often used to convert source programs from non-UNIX systems).
rjestat(I)	RJE status and enquiries.
send(I)	submit RJE job.

10. SOURCE CODE CONTROL SYSTEM (SCCS)

SCCS can be used to maintain, control, and identify files of text as they are modified and updated. Its most common use is for maintaining source programs, as well as for keeping track of successive versions of various documents; in combination with *diffmark*(I), this allows one to automatically generate "revision bars" in successive editions of such documents.

10.1 SCCS/PWB User's Manual ++

10.2 Manual Pages to Be Studied

Of the following, *get*(I), *delta*(I), and *prt*(I) are most frequently used.

admin(I)	administer SCCS files (including creation thereof).
chghist(I)	change the history entry of an SCCS delta.
comb(I)	combine SCCS deltas.
delta(I)	make an SCCS delta (a permanent record of editing changes).
get(I)	get a version of an SCCS file.
prt(I)	print SCCS file.

rmdel(I)	remove a delta from an SCCS file.
sccsdiff(I)	get the differences between two SCCS deltas.
what(I)	find and print SCCS identifications in files.

11. NUMERICAL COMPUTATION

11.1 DC—An Interactive Desk Calculator

11.2 BC—An Arbitrary Precision Desk Calculator Language

11.3 RATFOR—A Preprocessor for a Rational Fortran

11.4 Manual Pages to Be Studied

bas(I)	BASIC interpreter.
bc(I)	interactive language, acts as front end for *dc*(I)
dc(I)	desk calculator.
fc(I)	Fortran compiler/interpreter.
rc(I)	RATFOR preprocessor.

The PWB/UNIX* document entitled:

PWB/UNIX Beginner's Course

is not yet available.

* UNIX is a Trademark/Service Mark of the Bell System.

A Tutorial Introduction to the UNIX Text Editor

B. W. Kernighan

Bell Laboratories, Murray Hill, N. J.

Introduction

Ed is a "text editor", that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed*. The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the UNIX manual, all the while experimenting with *ed*. (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is.

You must also know what character to type as the end-of-line on your particular terminal. This is a "newline" on Model 37 Teletypes, and "return" on most others. Throughout, we will refer to this character, whatever it is, as "newline".

Getting Started

We'll assume that you have logged in to UNIX and it has just said "%". The easiest way to get *ed* is to type

```
ed        (followed by a newline)
```

You are now ready to go – *ed* is waiting for you to tell it what to do.

Creating Text – the Append command "a"

As our first problem, suppose we want to create some text starting from scratch. Perhaps we are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the 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 *ed* what to do to his text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – we will discuss these shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like "ready". (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append*, written as the letter

```
a
```

all by itself. It means "append (or add) text lines to the buffer, as I type them in." Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, we just type an "a" followed by a newline, followed by the lines of text we want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The "." is used to tell *ed* that we have finished appending. (Even experienced users forget that terminating "." sometimes. If *ed* seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The "a" and "." aren't there, because they are not text.

To add more text to what we already have, just issue another "a" command, and continue typing.

Error Messages — "?"

If at any time you make an error in the commands you type to *ed*, it will tell you by typing

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file — the Write command "w"

It's likely that we'll want to save our text for later use. To write out the contents of the buffer onto a file, we use the *write* command

```
w
```

followed by the filename we want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named "junk", for example, type

```
w junk
```

Leave a space between "w" and the file name. *Ed* will respond by printing the number of characters it wrote out. In our case, *ed* would respond with

```
68
```

(Remember that blanks and the newline character at the end of each line are included in the character count.) Writing a file just makes a copy of the text — the buffer's contents are not disturbed, so we can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a "w" command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed — the Quit command "q"

To terminate a session with *ed*, save the text you're working on by writing it onto a file using the "w" command, and then type the command

```
q
```

which stands for *quit*. The system will respond with "%". At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using "w". Then leave *ed* with the "q" command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to "%". Try both.)

Reading text from a file — the Edit command "e"

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the "w" command in a previous session. The *edit* command "e" fetches the entire contents of a file into the buffer. So if we had saved the three lines "Now is the time", etc., with a "w" command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file "junk" into the buffer, and respond

68

which is the number of characters in "junk". *If anything was already in the buffer, it is deleted first.*

If we use the "e" command to read a file into the buffer, then we need not use a file name after a subsequent "w" command; *ed* remembers the last file name used in an "e" command, and "w" will write on this file. Thus a common way to operate is

```
ed
e file
[editing session]
w
q
```

You can find out at any time what file name *ed* is remembering by typing the *file* command "f". In our case, if we typed

```
f
```

ed would reply

```
junk
```

Reading text from a file — the Read command "r"

Sometimes we want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command "r". The command

```
r junk
```

will read the file "junk" into the buffer; it adds it to the end of whatever is already in the buffer. So if we do a read after an edit:

```
e junk
r junk
```

the buffer will contain *two* copies of the text (six lines).

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the "w" and "e" commands, "r" prints the number of characters read in, after the reading operation is complete.

Generally speaking, "r" is much less used than "e".

Exercise 2:

Experiment with the "e" command — try reading and printing various files. You may get an error "?", typically because you spelled the file name wrong. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing the contents of the buffer — the Print command "p"

To *print* or list the contents of the buffer (or parts of it) on the terminal, we use the print command

```
p
```

The way this is done is as follows. We specify the lines where we want printing to begin and where we want it to end, separated by a comma, and followed by the letter "p". Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) we say

```
1,2p      (starting line=1, ending line=2 p)
```

Ed will respond with

```
Now is the time
for all good men
```

Suppose we want to print *all* the lines in the buffer. We could use "1,3p" as above if we knew there were exactly 3 lines in the buffer. But in general, we don't know how many there are, so what do we use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" — the dollar sign "$". Use it this way:

```
1,$p
```

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* will type

```
?
```

and wait for the next command.

To print the *last* line of the buffer, we could use

```
$,$p
```

but *ed* lets us abbreviate this to

```
$p
```

We can print any single line by typing the line number followed by a "p". Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, *ed* lets us abbreviate even further: we can print any single line by typing *just* the line number – no need to type the letter "p". So if we say

```
$
```

ed will print the last line of the buffer for us.

We can also use "$" in combinations like

```
$-1,$p
```

which prints the last two lines of the buffer. This helps when we want to see how far we got in typing.

Exercise 3:

As before, create some text using the append command and experiment with the "p" command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

```
3,1p
```

don't work.

The current line – "Dot" or "."

Suppose our buffer still contains the six lines as above, that we have just typed

```
1,3p
```

and *ed* has printed the three lines for us. Try typing just

```
p          (no line numbers).
```

This will print

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact it is the last (most recent) line that we have done anything with. (We just printed it!) We can repeat this "p" command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that we did anything to (in this case, line 3, which we just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

```
.          (pronounced "dot").
```

Dot is a line number in the same way that "$" is; it means exactly "the current line", or loosely, "the line we most recently did something to." We can use it in several ways – one possibility is to say

```
.,$p
```

This will print all the lines from (including) the current line to the end of the buffer. In our case these are lines 3 through 6.

Some commands change the value of dot, while others do not. The print command sets dot to the number of the last line printed; by our last command, we would have "." = "$" = 6.

Dot is most useful when used in combinations like this one:

```
.+1        (or equivalently, .+1p)
```

This means "print the next line" and gives us a handy way to step slowly through a buffer. We can also say

```
.-1        (or .-1p )
```

which means "print the line *before* the current line." This enables us to go backwards if we wish. Another useful one is something like

```
.-3,.-1p
```

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

```
.=
```

Ed will respond by printing the value of dot.

Let's summarize some things about the "p" command and dot. Essentially "p" can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter "p"), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single newline will cause printing of the next line – it's equivalent to ".+1p". Try it. Try typing "^" – it's equivalent to ".-1p".

Deleting lines: the "d" command

Suppose we want to get rid of the three extra lines in the buffer. This is done by the *delete* command

```
d
```

Except that "d" deletes lines instead of printing them, its action is similar to that of "p". The lines to be deleted are specified for "d" exactly as they are for "p":

starting line, ending line d

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left, as we can check by using

```
1,$p
```

And notice that "$" now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to "$".

Exercise 4:

Experiment with "a", "e", "r", "w", "p", and "d" until you are sure that you know what they do, and until you understand how dot, "$", and line numbers are used.

If you are adventurous, try using line numbers with "a", "r", and "w" as well. You will find that "a" will append lines *after* the line number that you specify (rather than after dot); that "r" reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that "w" will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

```
0r filename
```

and you can enter lines at the beginning of the buffer by saying

```
0a
... text ...
.
```

Notice that ".w" is *very* different from

```
.
w
```

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands – the substitute command

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. It is what we use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

```
Now is th time
```

– the "e" has been left off "the". We can use "s" to fix this up as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters 'th' the characters 'the'." To verify that it works (*ed* will not print the result automatically) we say

```
p
```

and get

```
Now is the time
```

which is what we wanted. Notice that dot must have been set to the line where the substitution took place, since the "p" command printed that line. Dot is always set this way with the "s" command.

The general way to use the substitute command is

starting-line, ending-line s/*change this*/*to this*/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between starting line and ending line. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for "p", except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error "?" as a warning.)

Thus we can say

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the "s" command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/something/something else/p

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, we can try again. (Notice that we put a print command on the same line as the substitute. With few exceptions, "p" can follow any command; no other multi-command lines are legal.)

It's also legal to say

s/ . . . //

which means "change the first string of characters to *nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if we had

Nowxx is the time

we can say

s/xx//p

to get

Now is the time

Notice that "//" here means "no characters", not a blank. There *is* a difference! (See below for another meaning of "//".)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

a
the other side of the coin
.
s/the/on the/p

You will get

on the other side of the coin

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a "g" (for "global") to the "s" command, like this:

s/ . . . / . . . /gp

Try other characters instead of slashes to delimit the two sets of characters in the "s" command – anything should work except blanks or tabs.

(If you get funny results using any of the characters

^ . $ [* \

read the section on "Special Characters".)

Context searching – "/ . . . /"

With the substitute command mastered, we can move on to another highly important idea of *ed* – context searching.

Suppose we have our original three line text in the buffer:

Now is the time
for all good men
to come to the aid of their party.

Suppose we want to find the line that contains "their" so we can change it to "the". Now with only three lines in the buffer, it's pretty easy to keep track of what line the word "their" is on. But if the buffer contained several hundred lines, and we'd been making changes, deleting and rearranging lines, and so on, we would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way we say "search for a line that contains this particular string of characters" is to type

/string of characters we want to find/

For example, the *ed* line

/their/

is a context search which is sufficient to find the desired line – it will locate the next occurrence of the characters between slashes ("their"). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

"Next occurrence" means that *ed* starts looking for the string at line ".+1", searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search "wraps around" from "$" to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, *ed* types the error message

?

Otherwise it prints the line it found.

We can do both the search for the desired line *and* a substitution all at once, like this:

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression "/their/" is a context search expression. In their simplest form, all context search expressions are like this – a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like "s". We used them both ways in the examples above.

Suppose the buffer contains the three familiar lines

Now is the time
for all good men
to come to the aid of their party.

Then the *ed* line numbers

/Now/+1
/good/
/party/−1

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, we could say

/Now/+1s/good/bad/

or

/good/s/good/bad/

or

/party/−1s/good/bad/

The choice is dictated only by convenience. We could print all three lines by, for instance

/Now/,/party/p

or

/Now/,/Now/+2p

or by any number of similar combinations. The first one of these might be better if we don't know how many lines are involved. (Of course, if there were only three lines in the buffer, we'd use

1,$p

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with "r", "w", and "a".)

Try context searching using "?text?" instead of "/text/". This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters – it's an easy way to back up.

(If you get funny results with any of the characters

^ . $ [* \

read the section on "Special Characters".)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

/string/

will find the next occurrence of "string". It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

//

This shorthand stands for "the most recently used context search expression." It can also be used as the first string of the substitute command, as in

/string1/s//string2/

which will find the next occurrence of "string1" and replace it by "string2". This can save a lot of typing. Similarly

??

means "scan backwards for the same expression."

Change and Insert – "c" and "i"

This section discusses the *change* command

c

which is used to change or replace a group of one or more lines, and the *insert* command

i

which is used for inserting a group of one or more lines.

"Change", written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines ".+1" through "$" to something else, type

.+1,$c
. . . type the lines of text you want here . . .
.

The lines you type between the "c" command and the "." will take the place of the original

lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the "c" command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of "." to end the input – this works just like the "." in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append – for instance

```
/string/i
. . . type the lines to be inserted here . . .
.
```

will insert the given text *before* the next line that contains "string". The text between "i" and "." is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

```
start, end d
i
. . . text . . .
.
```

is almost the same as

```
start, end c
. . . text . . .
.
```

These are not *precisely* the same if line "$" gets deleted. Check this out. What is dot?

Experiment with "a" and "i", to see that they are similar, but not the same. You will observe that

```
line-number a
. . . text . . .
.
```

appends *after* the given line, while

```
line-number i
. . . text . . .
.
```

inserts *before* it. Observe that if no line number is given, "i" inserts before line dot, while "a" appends after line dot.

Moving text around: the "m" command

The move command "m" is used for cutting and pasting – it lets you move a group of lines from one place to another in the buffer. Suppose we want to put the first three lines of the buffer at the end instead. We could do it by saying:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) but we can do it a lot easier with the "m" command:

```
1,3m$
```

The general case is

start line, end line m *after this line*

Notice that there is a third line to be specified – the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if we had

```
First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.
```

we could reverse the two paragraphs like this:

```
/Second/,/second/m/First/-1
```

Notice the "-1" – the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands "g" and "v"

The *global* command "g" is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain "peling". More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the "g" command does not give a "?" if "peling" is not found where the "s" command will.

There may be several commands (including "a", "c" "i" "r", "w", but not "g"); in that case, every line except the last must end with a backslash "\":

```
g/xxx/.-1s/abc/def/\
.+2s/ghi/jkl/\
.-2,.p
```

makes changes in the lines before and after each line that contains "xxx", then prints all three lines.

The "v" command is the same as "g", except that the commands are executed on every line that does *not* match the string following "v":

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like ".", "*", "$", and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only,*

```
/x.y/
```

means "a line with an x, *any character,* and a y," *not* just "a line with an x, a period, and a y." A complete list of the special characters that can cause trouble is the following:

```
^  .  $  [  *  \
```

Warning: The backslash character \ is special to *ed.* For safety's sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change "\.*" into "backslash dot star".

Here is a hurried synopsis of the other special characters. First, the circumflex " ^ " signifies the beginning of a line. Thus

```
/^string/
```

finds "string" only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign "$" is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of "string" that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just "string", and

```
/^.$/
```

finds a line containing exactly one character.

The character ".", as we mentioned above, matches anything;

```
/x.y/
```

matches any of

```
x+y
x-y
x y
x.y
```

This is useful in conjunction with "*", which is a repetition character; "a*" is a shorthand for "any number of a's," so ".*" matches any number of anythings. This is used like this:

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since ".*" finds the longest possible match, this goes up to the last comma.)

"[" is used with "]" to form "character classes"; for example,

```
/[1234567890]/
```

matches any single digit — any one of the characters inside the braces will cause a match.

Finally, the "&" is another shorthand character - it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing. Suppose the current line contained

```
Now is the time
```

and we wanted to put parentheses around it. We could just retype the line, but this is tedious. Or we could say

```
s/^/(/
s/$/)/
```

using our knowledge of "^" and "$". But the easiest way uses the "&":

```
s/.*/(&)/
```

This says "match the whole line, and replace it by itself surrounded by parens." The "&" can be used several times in a line; consider using

```
s/.*/&? &!!/
```

to produce

Now is the time? Now is the time!!

We don't have to match the whole line, of course: if the buffer contains

the end of the world

we could type

/world/s//& is at hand/

to produce

the end of the world is at hand

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string "/world/" found the desired line; the shorthand "//" found the same word in the line; and the "&" saved us from typing it again.

The "&" is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. We can turn off the special meaning of "&" by preceding it with a "\":

s/ampersand/\&/

will convert the word "ampersand" into the literal symbol "&" in the current line.

Summary of Commands and Line Numbers

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of *e*, *r* and *w*, followed by a file name. Only one command is allowed per line, but a *p* command may follow any other command (except for *e*, *r*, *w* and *q*).

a (append) Add lines to the buffer (at line dot, unless a different line is specified). Appending continues until "." is typed on a new line. Dot is set to the last line appended.

c (change) Change the specified lines to the new text which follows. The new lines are terminated by a ".". If no lines are specified, replace line dot. Dot is set to last line changed.

d (delete) Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless "$" is deleted, in which case dot is set to "$".

e (edit) Edit new file. Any previous contents of the buffer are thrown away, so issue a *w* beforehand if you want to save them.

f (file) Print remembered filename. If a name follows *f* the remembered name will be set to it.

g (global) g/---/commands will execute the commands on those lines that contain "---", which can be any context search expression.

i (insert) Insert lines before specified line (or dot) until a "." is typed on a new line. Dot is set to last line inserted.

m (move) Move lines specified to after the line named after *m*. Dot is set to the last line moved.

p (print) Print specified lines. If none specified, print line dot. A single line number is equivalent to "line-number p". A single newline prints ".+1", the next line.

q (quit) Exit from ed. Wipes out all text in buffer!!

r (read) Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

s (substitute) s/string1/string2/ will substitute the characters of 'string2' for 'string1' in specified lines. If no line is specified, make substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. *s* changes only the first occurrence of string1 on a line; to change all of them, type a "g" after the final slash.

v (exclude) v/---/commands executes "commands" on those lines that *do not* contain "---".

w (write) Write out buffer onto a file. Dot is not changed.

.= (dot value) Print value of dot. ("=" by itself prints the value of "$".)

! (temporary escape)
Execute this line as a UNIX command.

/---/ Context search. Search for next line which contains this string of characters. Print it. Dot is set to line where string found. Search starts at ".+1", wraps around from "$" to 1, and continues to dot, if necessary.

?---? Context search in reverse direction. Start search at ".−1", scan to 1, wrap around to "$".

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help secretaries, typists and programmers to make effective use of the UNIX facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor **ed**;
- commands for "cut and paste" operations on files and parts of files, including the **mv**, **cp**, **cat** and **rm** commands, and the **r**, **w**, **m** and **t** commands of the editor;
- editing scripts and editor-based programs like **grep** and **sed**.

Although the treatment is aimed at non-programmers, new UNIX users with any background should find helpful hints on how to get their jobs done more easily.

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Although UNIX provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might.

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in *The UNIX Programmer's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on **ed**, like **grep** and **sed**.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor **ed** is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of **ed** for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

The List command 'l'

ed provides two commands for printing the contents of the lines you're editing. Most people are familiar with **p**, in combinations like

```
1,$p
```

to print all the lines you're editing, or

```
s/abc/def/p
```

to change 'abc' to 'def' on the current line. Less familiar is the *list* command **l** (the letter 'l'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will print each tab as ➢ and each backspace as ≺. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \, so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the **l** command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command **s**. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any **ed** command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing **g** after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing **g** changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to 'print' or 'list' (as described in the previous section) the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all legal, and mean slightly different things. Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a **p** or **l** to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command **u** lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter '.'

As you have undoubtedly noticed when you use **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period '.'. On the left side of a substitute command, or in a search with '/.../', '.' stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where 'x' and 'y' occur separated by a single character, as in

```
x+y
x−y
x□y
x.y
```

and so on. (We will use □ to stand for a space whenever we need to make it clear.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by **l**. Suppose you have a line that, when printed with the **l** command, appears as

```
.... th\07is ....
```

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

```
s/\07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is.*

Bear in mind that since '.' matches any single character, the command

```
s/./,/
```

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in **ed**, the '.' has several meanings, depending on its context. This line shows all three:

```
.s/./,/
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The

second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

Now is the time.

the result will be

.ow is the time.

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

Now is the time.

into

Now is the time?

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

Now is the time.

like this:

s/\./?/

The pair of characters '\.' is considered by ed to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

.PP

The search

/.PP/

isn't adequate, for it will find a line like

THE APPLICATION OF ...

because the '.' matches the letter 'A'. But if you say

/\.PP/

you will find only lines that contain '.PP'.

The backslash can also be used to turn off special meanings for characters other than '.'. For example, consider finding a line that contains a backslash. The search

/\/

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

/\\/

does work. Similarly, you can search for a forward slash '/' with

/\//

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line

\x\.\y

into the line

\x\y

Here are several solutions; verify that each works as advertised.

s/\\\./
s/x../x/
s/..y/y/

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an s command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

//exec //sys.fort.go // etc...

you could use a colon as the delimiter — to delete all the slashes, type

s:/::g

Second, if # and @ are your character erase and line kill characters, you have to type \# and \@; this is true whether you're talking to ed or any other program.

When you are adding text with a or i or c, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '$'

The next metacharacter, the '$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

Now is the

and you wish to add the word 'time' to the end. Use the $ like this:

s/$/␣time/

to get

Now is the time

Notice that a space is needed before 'time' in the substitute command, or you will get

Now is thetime

As another example, replace the second comma in this line with a period without altering the first.

Now is the time, for all good men,

The command needed is

s/,$/./

The $ sign here provides context to make specific which comma we mean. Without it, of course, the s command would operate on the first comma to produce

Now is the time. for all good men,

As another example, to convert

Now is the time.

into

Now is the time?

as we did earlier, we can use

s/.$/?/

Like '.', the '$' has multiple meanings depending on context. In the line

$s/$/$/

the first '$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

/the/

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

/^the/

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

s/^/␣/

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

.PP

you can use the command

/^\.PP$/

The Star '*'

Suppose you have a line that looks like this:

text x y *text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the x and the y. Suppose the job is to replace all the spaces between x and y by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

s/x␣*y/x␣y/

The construction '␣*' means 'as many spaces as possible'. Thus 'x␣*y' means 'an x, as many spaces as possible, then a y'.

The star can be used with any character, not just space. If the original example was instead

text x---------y *text*

then all '-' signs can be replaced by a single space with the command

s/x-*y/x␣y/

Finally, suppose that the line was

text x.....y *text*

Can you see what trap lies in wait for the unwary? If you blindly type

s/x.*y/x□y/

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x *text* x.....y *text* y *text*

then saying

s/x.*y/x□y/

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

s/x\.*y/x□y/

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.*' is exactly what you want. For example, to change

Now is the time for all good men ....

into

Now is the time.

use '.*' to eat up everything after the 'for':

s/ for.*/./

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

text xy *text* x y *text*

and we said

s/x□*y/x□y/

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

/x□□*y/

which says 'an x, a space, then as many more spaces as possible, then a y'.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

s/x*/y/g

when applied to the line

abcdef

produces

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

s/xx*/y/g

'xx*' is one or more x's.

The Brackets '[ ]'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

[1234567890]

matches any single digit — the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so

1,$s/^[1234567890]*//

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for

example, you can say

```
/[.\$^[]/
```

Within [...], the '[' is not special. To get a ']' into a character class, make it the first character.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

```
[^1234567890]
```

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^^]/
```

finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

```
s/the/the best/
```

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

```
s/the/& best/
```

and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

```
s/ampersand/\&/
```

converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

```
text  xy  text
```

you can break it between the 'x' and the 'y' like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the **roff** or **nroff** formatting command '.ul'.

```
text a very big text
```

The command

```
s/␣very␣/\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

Regrettably there is no way to go in the opposite direction: ed will not convert two lines into one.

Rearranging a Line with \(... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an s command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '\1' refers to whatever matched the first \(...\) pair, '\2' to the second \(...\), and so on.

The command

```
1,$s/^\([^,]*\),□*\(.*\)/\2□\1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with '\1'. The second \(...\) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands g and v discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in ed, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

```
1,$s/x/y/
```

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '.', '$', '/.../' and '?...?' with '+' and '−'. Thus

```
$−1
```

is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session,

```
$−5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

```
.−3,.+3p
```

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

```
.−3,.3p
```

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '−' and '+' as line numbers by themselves.

```
-
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

```
---
```

moves up three lines, as does '−3'. Thus

```
-3,+3p
```

is also identical to the examples above.

Since '−' is shorter than '.−1', constructions like

```
-,.s/bad/good/
```

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '−' can be used in combination with searches using '/.../' and '?...?', and with '$'. The search

```
/thing/--
```

finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

```
/horrible thing/
.... ed prints line with 'horrible thing' ...
s//good/p
```

To go backwards and change a line, say

```
??s//good/
```

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

```
//s//&□&/p
```

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the *new* line '$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
... text ...
... botch ...                (minor error)
.
s/botch/correct/             (fix botched line)
a
... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
... text ...
... horrible botch ...          (major error)
.
c                               (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **0r** to read a file in at the beginning of the text. (You can also say **0a** or **1i** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the **s** command. The rule is simple — you are left sitting on the last line that matched the pattern. If there were no matches, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In **ed**, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

```
/a/;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing/;//
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while **ed** is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the **p** command was started.

4. GLOBAL COMMANDS

The global commands **g** and **v** are used to perform one or more editing commands on all lines that either contain (**g**) or don't contain (**v**) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word UNIX. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with '.').

The **v** command is identical to **g**, except that it operates on those line that do *not* contain an occurrence of the pattern. (There is no mnemonic significance to the letter 'v'.) So

```
v/^\./p
```

prints all the lines that don't begin with '.' — the actual text lines.

The command that follows **g** or **v** can be anything:

```
g/^\./d
```

deletes all lines that begin with '.', and

```
g/^$/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word UNIX to 'Unix' everywhere, and verify that it really worked, with

```
g/UNIX/s//Unix/gp
```

Notice that we used '//' in the substitute command to mean 'the previous pattern', in this case, UNIX. The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, and so on quite freely.

```
g/^\.PP/+
```

prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between '.EQ' and '.EN' formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The '\' signals the **g** command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a **g** command.)

You should watch out for this problem: the command

```
g/x/s//y/\
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a '.' to terminate the **i** command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss **ed** commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called **mv** (for 'move'); it 'moves' the file from one name to another, like this:

```
mv memo paper
```

That's all there is to it: **mv** from the old name to the new name.

```
mv oldname newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv x x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just

because you're paranoid.

In any case, the way to do it is with the **cp** command. (**cp** stands for 'copy'; UNIX is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'savegood' might be acceptable — then type

```
cp good savegood
```

This copies 'good' onto 'savegood', and you now have two identical copies of the file 'good'. (If 'savegood' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv savegood good
```

(if you're not interested in 'savegood' anymore), or

```
cp savegood good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm savegood
```

throws away (irrevocably) the file called 'savegood'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called **cat**. (Not *all* UNIX programs have two-letter names.) **cat** is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'bigfile'. If you say

```
cat file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat file1 file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'bigfile'.

Fortunately, there is a way. You can tell UNIX that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character > and the name of the file where you want the output to go. Then you can say

```
cat file1 file2 >bigfile
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'bigfile', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of UNIX. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat file1 file2 file3 ... >bigfile
```

collects a whole bunch.

Question: is there any difference between

```
cp good savegood
```

and

```
cat good >savegood
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat good good1 >temp
mv temp good
```

which is probably most direct. You should also understand why

```
cat good good1 >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the **ed** commands for reading and writing files. Of course you can't go very far without knowing **r** and **w**. Equally useful, but less well known, is the 'edit' command **e**. Within **ed**, the command

```
e newfile
```

says 'I want to edit a new file called *newfile*, without leaving the editor.' The **e** command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the **q** command, then re-entered **ed** with a new file name, except that if you have a pattern remembered, then a command like // will still work.

If you enter **ed** with the command

```
ed file
```

ed remembers the name of the file, and any subsequent **e**, **r** or **w** commands that don't contain a filename will refer to this remembered file. Thus

```
ed file1
... (editing) ...
w       (writes back in file1)
e file2  (edit new file, without leaving editor)
... (editing on file2) ...
w       (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving **ed** and without typing the name of any file more than once.

You can find out the remembered file name at any time with the **f** command; just type **f** without a file name. You can also change the name of the remembered file name with **f**; a useful sequence is

```
ed precious
f junk
... (editing) ...
```

which gets a copy of a precious file, then uses **f** to guarantee that a careless **w** command won't clobber the original.

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by **nroff** or **troff**. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

```
ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r table
```

The critical line is the last one. As we said earlier, the **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command without any address adds lines at the end, so it is the same as **$r**.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
...[lots of stuff]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```
/^\.TS/
.TS [ed prints the line it found]
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```
a
...lots of stuff...
...horrible line...
.
.w  temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```
.,/^\.PP/-w temp
.,//-d
$r temp
```

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/-') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command **m** that **ed** provides — it lets you do the whole set of operations at one crack, without any temporary file.

The **m** command is like many other **ed** commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, $ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
.,/^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot.

As you can see, the **m** command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched **m** command can often be a mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a **w** command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line

with the address

'x

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *'a*. Then find the last line and mark it with *'b*. Now position yourself at the place where the stuff is to go and say

'a,'bm.

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called t (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier that writing and reading.

The t command is identical to the m command, except that instead of moving lines it simply duplicates them at the place you named. Thus

1,t

duplicates the entire contents that you are editing. A more common use for t is for creating a series of lines that differ only slightly. For example, you can say

```
a
..... x ..... (long line)
.
t.              (make a copy)
s/x/y/          (change it a bit)
t.              (make third copy)
s/y/z/          (change it a bit)
```

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command ! provides a way to do this.

If you say

!any UNIX command

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another !; at that point you can resume editing.

You can really do *any* UNIX command, including another **ed**. (This is quite common, in fact.) In this case, you can even do another !.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how **ed** works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are big (more than three or four thousand lines) it is impossible because of limits in **ed**.

The program **grep** was invented to get around these limitations. The patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

g/re/p

That describes exactly what **grep** does — it prints every line in a set of files that contains a particular pattern. Thus

grep 'thing' file1 file2 file3 ...

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. **grep** also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since **grep** and **ed** use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before **grep** gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep −v 'thing' file1 file2 ...
```

finds all lines that don't contains 'thing'. The **−v** must occur in the position shown. Given **grep** and **grep −v**, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep −v y
```

(The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every UNIX to Unix and every GCOS to Gcos in a large number of files. Then put into the file 'script' the lines

```
g/UNIX/s//Unix/g
g/GCOS/s//Gcos/g
w
q
```

Now you can say

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically **sed** copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the UNIX to Unix part of the example given above, but without rewriting the files. Then the command

```
sed 's/UNIX/Unix/g' file1 file2 ...
```

applies the command 's/UNIX/Unix/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using **sed** in such a case is that it can be used with input too large for **ed** to handle. and that all the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed −f cmdfile input−files...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

Acknowledgement

I am grateful to Ted Dolotta for his careful reading and valuable suggestions.

References

[1] Brian W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor*, Bell Laboratories internal memorandum.

[2] Brian W. Kernighan, *UNIX For Beginners*, Bell Laboratories internal memorandum.

[3] Ken L. Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual*. Bell Laboratories, 1975.

PWB/UNIX

Shell Tutorial

J. R. Mashey

September 1977

Bell Telephone Laboratories, Incorporated

PWB/UNIX Shell Tutorial

CONTENTS

PWB/UNIX Shell Tutorial

J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

The command language for PWB/UNIX* is a high-level programming language that is an extended version of the UNIX Shell. By utilizing the Shell as a programming language, one can eliminate much of the programming drudgery that often accompanies a large project. Many manual procedures can be quickly, cheaply, and conveniently automated. Because it is so easy to create and use Shell procedures, individual users and entire projects can customize the general PWB/UNIX environment into one tailored to their own respective requirements, organizational structure, and terminology.

This paper is actually a combination of several tutorials, as explained in {1}.[1] Some sections provide a basic tutorial for relatively new users. Other sections are intended for more experienced users and introduce them to Shell programming. Finally, some hints on programming techniques and efficiency are offered for those who make especially heavy use of Shell programming.

The accuracy of this tutorial is guaranteed *only* for the Shell of PWB/UNIX—Edition 1.0. Other versions of UNIX have other Shells. Although many of the basic concepts are similar, there exist many differences in features, especially those used to support Shell programming.

1. INTRODUCTION

In any programming project, some effort is used to build the end product. The remainder is consumed in building the supporting tools and procedures used to manage and maintain the end product. The second effort can far exceed the first, especially in larger projects. A good command language can be an invaluable tool for such projects. If it is a flexible programming language, it can be used to solve many internal support problems, without requiring compilable programs to be written, debugged, and maintained; its most important advantage is the ability to get the job done *now*. For a perspective on the motivations for using a command language in this way, see [1,2,6].

When users log into a PWB/UNIX system, they communicate with an instance of the Shell that reads commands typed at the terminal and arranges for their execution. Thus, the Shell's most important function is to provide a good interface for human beings. In addition, a sequence of commands may be preserved for later use by saving them in a file, called a *Shell procedure*, a *command file*, or a *runcom*, according to local preferences.

Some users need little knowledge of the Shell to do their work; others choose to make heavy use of its programming features. This tutorial may be read in several different ways, depending on the reader's interests. A brief discussion of the PWB/UNIX environment is found in {2}. The discussion in {3} covers aspects of the Shell that are important for everyone, while all of {4} and most of {5} are mainly of interest to those who write Shell procedures. A group of annotated Shell procedures is given in {6}. Finally, a brief discussion of efficiency is offered in {7}. This is found in its proper place (the end), and is intended for those who write especially time-consuming Shell procedures.

Complete beginners should *not* be reading this tutorial, but should work their way through other available tutorials first. See [7] for an appropriate plan of study. All the *commands* mentioned below are described in Section I of the *PWB/UNIX User's Manual* [3], while *system calls* are described in Section II and *subroutines* in Section III thereof.

2. OVERVIEW OF THE UNIX ENVIRONMENT

Full understanding of some later discussions depends on familiarity with PWB/UNIX; [9] is most useful for that, and it would be helpful to read at least one of [4,5,10]. For completeness, a short overview of the most relevant concepts is given below.

---

* UNIX is a Trademark/Service Mark of the Bell System.

1. The notation {*n*} refers to Section *n* of this tutorial.

2.1 File System

The PWB/UNIX file system's overall structure is that of a rooted tree composed of *directories* and other files. A *file name* is a sequence of characters. A *pathname* is a sequence of directory names followed by a file name, each separated from the previous one by a slash (/). If a pathname begins with a "/", the search for the file begins at the root of the entire tree; otherwise, it begins at the user's *current directory* (also known as the *working directory*). (The first kind of name is often called a *full pathname* because it is invariant with regard to the user's current directory.) The user may change the current directory at any time by using the *cd* or *chdir* command. In most cases, a file name and its corresponding pathname may be used interchangeably. Some sample names are:

.	name of the current directory.
..	name of the parent directory of the current directory.
/	root directory of the entire file structure.
/bin	directory containing most of the frequently-used public commands.
/a1/tf/jtb/bin	a full pathname typical of multi-person programming projects. This one happens to be a private directory of commands belonging to person "jtb" in project "tf"; "a1" is the name of a *file system.*
bin/umail	a name depending on the current directory: it names file "umail" in subdirectory "bin" of the current directory. If the current directory is "/", it names "/bin/umail". If the current directory is "/a1/tf/jtb", it names "/a1/tf/jtb/bin/umail".
memox	name of a file in the current directory.

2.2 Processes

☛ *Beginners should skip this section on first reading.*

An *image* is a computer execution environment, including memory image, register values, current directory, status of open files, information recorded at login time, and various other items. A *process* is the execution of an image; most PWB/UNIX commands execute as separate processes. One process may spawn another using the *fork* system call, which duplicates the image of the original (*parent*) process. The new (*child*) process continues to execute the same program as the parent. The two images are identical, except that the program can determine whether it is executing as parent or child. The program may continue execution of the image or may abandon it by issuing an *exec* system call, thus initiating execution of another program. In any case, each process is free to proceed in parallel with the other, although the parent quite commonly issues a *wait* system call to suspend execution until a child exits.

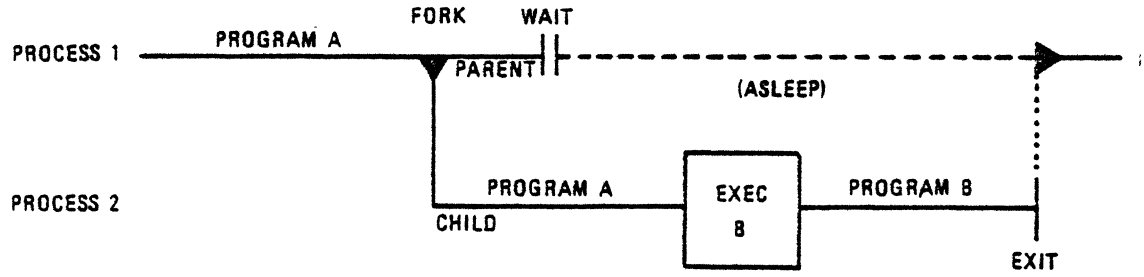


Figure 1

Figure 1 illustrates these ideas. *Program A* is executing (as *process 1*) and wishes to run *program B*. It "forks" and spawns a child (*process 2*) that continues to execute *program A*. The child abandons *A* by executing *B*, while the parent goes to sleep until the child exits.

A child inherits its parent's *open files*. This mechanism permits processes to share a common input stream in various ways. In particular, an open file possesses a *pointer* that indicates a position in the file and is modified by various operations. *Read* and *write* system calls copy a requested number of bytes from or to a file, beginning at the position given by the current value of the pointer. As a side effect, the pointer is incremented by the number of bytes transferred, yielding the effect of sequential I/O. *Seek* can be used to obtain random-access I/O; it sets the pointer to an absolute position within the file, or to a position offset either from the end of the file or from the current pointer position.

When a process terminates, it can set an eight-bit *return code* (or *exit code*) that is available to its parent. This code is usually used to indicate success or failure.

Signals indicate the occurrence of events that may have some impact on a process. A signal may be sent to a process by another process, from the keyboard, or by PWB/UNIX itself. For most types of signals, a process can arrange to be terminated on receipt of a signal, to ignore it completely, or to "catch" it and take appropriate action [4.6]. For example, an *interrupt* signal may be sent by depressing an appropriate key ("del", "break", or "rubout"). The action taken depends on the requirements of the specific program being executed:

- The Shell invokes most commands in such a way that they immediately die when an interrupt is received. For example, *pr* normally dies, allowing the user to terminate unwanted output.
- The Shell itself ignores interrupts when reading from the terminal, because it should continue execution even when the user terminates a command like *pr*.
- The editor *ed* chooses to "catch" interrupts so that it can halt its current action (especially printing) without terminating completely.

Limiting interprocess communication to a small number of well-defined methods is a great aid to uniformity, understandability, and reliability of programs. It encourages the "packaging" of each function into a small program that is easily connected to other programs, but depends very little on the internal workings of other programs.

3. SHELL BASICS

The *Shell* (i.e., the *sh* command) implements the command language visible to most PWB/UNIX users. It reads input from a terminal or a file and arranges for the execution of the requested commands. It is a small program (about forty pages of C code); many of its functions are actually provided by independent programs that work with it. It is *not* part of the operating system, but is an ordinary user program. The discussion below is adapted from [10,11].

3.1 Commands

A *command* is a sequence of non-blank arguments separated by blanks or tabs. The first argument (numbered *zero*) specifies the name of the command to be executed; any remaining arguments are passed as character-strings to the command executed. A command may be as simple as:

```
who
```

which prints the login names of logged-in users. The following line requests the *pr* command to print files a, b, and c:

```
pr a b c
```

If the first argument names a file that is *executable*[2] and is actually a load module, the Shell (as parent) spawns a new (child) process that immediately executes that program. If the file is marked executable, but is neither a load module nor a directory, it is assumed to be a *Shell procedure*, i.e., a file of ordinary text containing Shell command lines and possibly lines to be read by other programs. In this case, the Shell spawns a new instance of itself to read the file and execute the commands included in it. The following command requests that the on-line *PWB/UNIX User's Manual* [3] pages for the *who* and *pr* commands be printed on the terminal (the *man* command is actually implemented as a Shell procedure):

```
man who pr
```

2. As evidenced by an appropriate set of permission bits associated with that file.

From the user's viewpoint, executable programs and Shell procedures are invoked in exactly the same way. The Shell determines which implementation has been used, rather than requiring the user to do so. This preserves the uniformity of invocation and the ease of changing the implementation choice for a given command. The actions of the Shell in executing any of these commands are illustrated in Figure 1 {2.2}.

3.2 Redirection of Standard Input and Output

When a command begins execution, it usually expects that three files are already open, a "standard input", a "standard output", and a "diagnostic output". When the user's original Shell is started, all three have already been opened to the user's terminal. A child process normally inherits these files from its parent. The Shell permits them to be redirected elsewhere before control is passed to an invoked command.

An argument to the Shell of the form "<file" or ">file" opens the specified file as standard input or output, respectively. An argument of the form ">>file" opens the standard output to the end of the file, thus providing a way to append data to it. In either output case, the Shell creates the file if it did not already exist. The following appends to file "log" the list of users who are logged in:

```
who >>log
```

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal or file. Thus, commands can be used conveniently in many different contexts. A few commands vary their actions depending on the nature of their input or output, either for efficiency's sake, or to avoid useless actions (such as attempting random-access I/O on a terminal).

Redirection of the diagnostic output is discussed in {4.7.3}.

3.3 Command Lines

A sequence of commands separated by "|" (or "^") make up a *pipeline.* Each command is run as a separate process connected to its neighbor(s) by *pipes,* i.e., the output of each command (except the last one) becomes the input of the next command in line. A *filter* is a command that reads its input, transforms it in some way, then writes it as output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, they are synchronized to the extent that each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it, and looping back for more input. Some must read larger amounts of data before producing output; *sort* is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline: *nroff* is a text formatter whose output may contain reverse line motions; *col* converts these motions to a form that can be printed on a terminal lacking reverse motion capability; *reform* is used here to speed printing by converting the (tab-less) output of *col* to an equivalent one containing horizontal tab characters. The flag "−mm" indicates one of the more-commonly used formatting options, and "text" is the name of the file to be formatted:

```
nroff −mm text | col | reform
```

Figure 2 shows the sequence of actions that set up this pipeline. Not shown are actions by the Shell that create pipes and manipulate open files, causing the commands to be tied together correctly.

A *command line* consists of zero or more pipelines separated by semicolons or ampersands. If the last command in a pipeline is terminated by a semicolon (;) or a new-line character, the Shell waits for the command to finish before continuing to read command lines. It does *not* wait if the pipeline is terminated by an ampersand (&); both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily, or until its processes are *killed.* The first example below executes *who,* waits for it to terminate, and then executes *date;* the second invokes both commands in order, but does not wait for either one to finish. Figure 3 shows the actions of the Shell involved in executing these examples:

```
who >log; date
who >log& date&
```

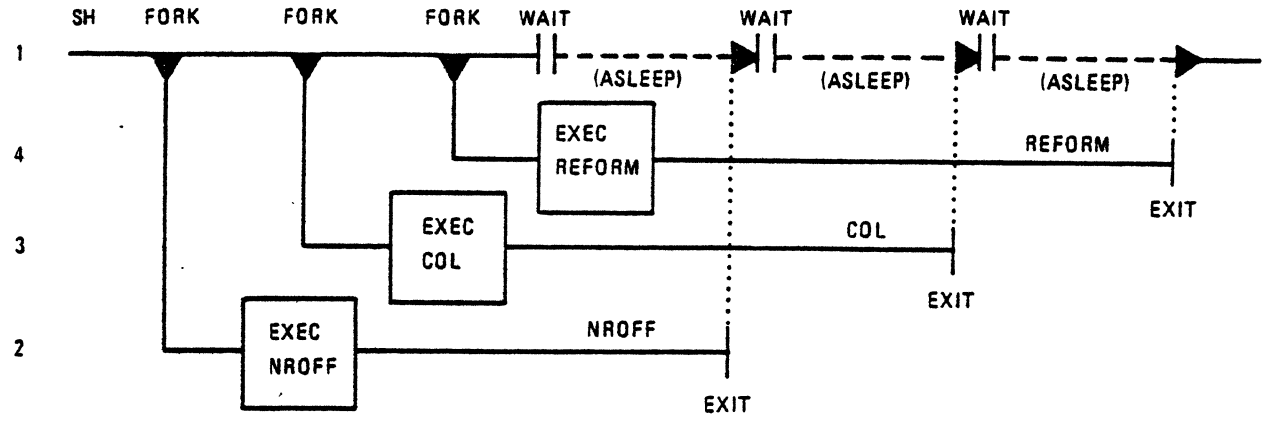


Figure 2

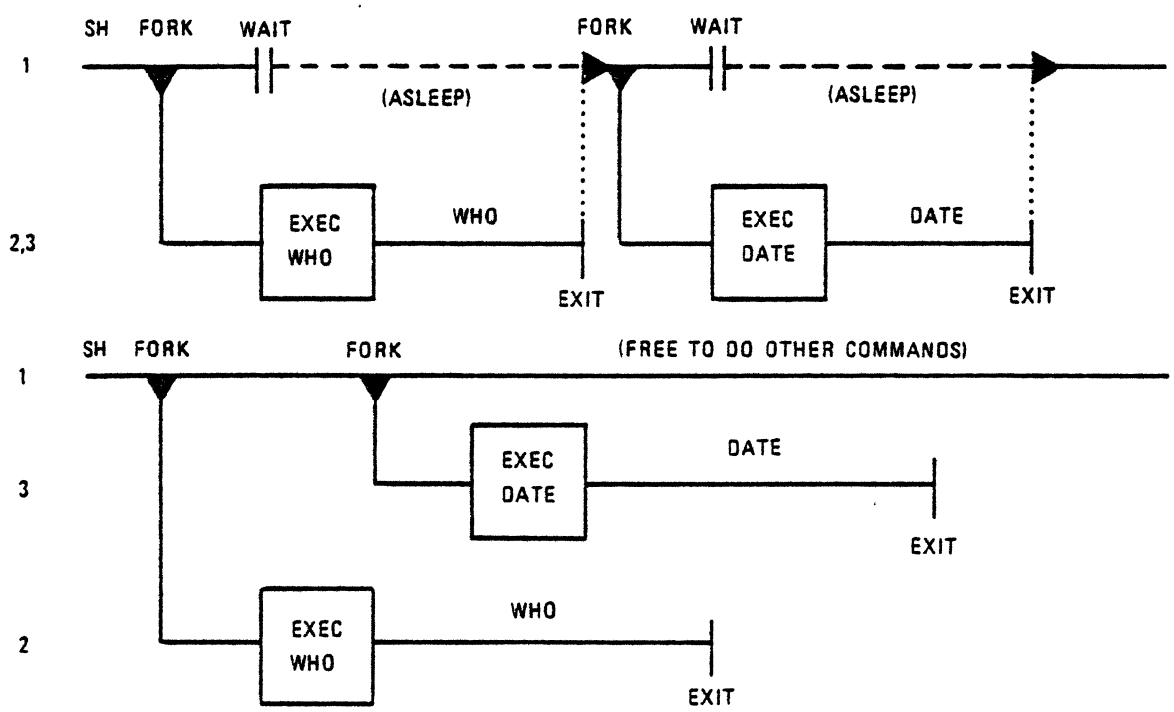


Figure 3

More typical uses of "&" include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example:

```
nohup cc prog.c&
```
You continue working while the C compiler runs in background.

A command terminated by "&" is immune to interrupts, but it is wise to make it immune to hang-ups as well. The *nohup* command is used for this purpose. Without *nohup,* if you hang up while *cc* (the C compiler) is still executing, *cc* will be killed and your output will disappear.

☛ *The "&" operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of simultaneous, asynchronous processes without a compelling reason for doing so.*

A simple command in a pipeline may be replaced by a command line enclosed in parentheses "()"; in this case, another instance of the Shell is spawned to execute the commands so enclosed. This action is

helpful in combining the output of several sequentially executed commands into a stream to be processed by a pipeline. The following line prints two separate documents in a way similar to that shown in a previous example:

```
(nroff -mm text1; nroff -mm text2) | col | reform
```

3.4 Generation of Argument Lists

Many command arguments are names of files. When certain characters are found in an argument, they cause replacement of that argument by a sorted list of zero or more file names obtained by pattern-matching on the contents of a directory. Most characters match themselves. The "?" matches *any one character;* the "*" matches *any string* of any characters (other than "/"), including the null string. Enclosing a set of characters within square brackets "[...]" causes the construct to match *any one character* in that set.[3] Inside square brackets, a pair of characters separated by "-" includes in the set all characters lexically within the inclusive range of that pair.

For example, "*" matches all files in the current directory, "*tmp*" matches all names containing "tmp", "[a-f]*" matches all files whose names begin with "a" through "f", "*.c" matches all files ending in ".c", and "/a1/tf/bin/?" matches all single-character names found in "/a1/tf/bin". This capability saves much typing, and more importantly, makes it possible to organize information as large collections of small files that are named in disciplined ways.

Pattern-matching has several restrictions. If the first character of a file name is ".", it can be matched only by an argument that begins with ".". Pattern-matching is currently restricted to the last component in a pathname—the string "/a1/tf/*" is legal, but the string "/a1/*/bin" is not. Pattern-matching does not apply to the name of the invoked command (i.e., argument number 0).

3.5 Quoting Mechanisms

If a character has a special meaning to the Shell, that meaning may be removed by preceding the character with a back-slash (\); the "\" acts as an escape and disappears. A "\" followed by a new-line character is treated as a blank, permitting continuation of commands on additional input lines. A sequence of characters enclosed in single quotes ('...') is taken literally—"what you see is what you get". The beginner should use single quotes in most instances. Double quotes ("...") are required in a few cases, primarily inside Shell procedures. Double quotes hide the significance of most special characters, but allow substitution of Shell arguments and variables; see {4.8} for further details.

3.6 Examples

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these examples at a terminal:

- who
  Print (on the terminal) the list of logged-in users.
- who >>log
  Append the list of logged-in users to the end of file "log".
- who | wc -l
  Print the number of logged-in users. (The argument to *wc* is "minus ell".)
- who | pr
  Print a paginated list of logged-in users.
- who | sort
  Print an alphabetized list of logged-in users.
- who | grep pw
  Print the list of logged-in users whose login names contain "pw".
- who | grep pw | sort | pr
  Print an alphabetized, paginated list of logged-in users whose names contain "pw".

3. Be warned that square brackets are also used below in an entirely different sense: in descriptions of commands, they indicate that the enclosed argument is optional.

- (date; who | wc -l) >>log
  Append (to "log") the current date followed by the count of logged-in users.
- who | sed 's/ .*//' | sort | uniq -d
  Print only the login names of all users who are logged in more than once.

The *who* command does not *by itself* provide options to yield all these results—they are obtained by combining it with other commands. The kinds of operations illustrated above may be used in other circumstances; *who* just serves as the data source in these examples. As an exercise, replace "who |" by "</etc/passwd" in the above examples to see how a file can be used as a data source in the same way.

3.7 How the Shell Finds Commands

The Shell normally searches for commands in a way that permits them to be found in three distinct locations in the file structure. The Shell first attempts to use the command name as given; if this fails, it prepends the string "/bin/" to the name, and, finally, "/usr/bin/". The effect is to search, in order, the current directory, "/bin", and "/usr/bin". For example, the *pr* and *man* commands are actually located in files "/bin/pr" and "/usr/bin/man", respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command via an absolute pathname. If a command name *as given* contains a "/" (e.g., "/bin/sort" or "../cmd"), the prepending is *not* performed. Instead, a single attempt is made to execute the unmodified command name.

This mechanism gives the user a convenient way to execute public commands and commands in or "near" the current directory, as well as the ability to execute *any* accessible command regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without interfering with other users. Similarly, the creation of a new public command will not affect a user who already has a private command with the same name. This mechanism may be overridden {4.4}.

3.8 Changing the State of the Shell and the .profile File

The state of a given instance of the Shell may be altered in various ways. The following commands are used more often at the terminal than in Shell procedures.

The *cd* command (or its synonym *chdir*) changes the current directory of the Shell to the one specified. This can (and should) be used to change to a convenient place in the directory structure; *cd* is often combined with "()" to cause a sub-Shell to change to a different directory and execute some commands, without affecting the original Shell. The first sequence below extracts the component files of the archive file "/a1/tf/q.a" and places them in whatever directory is the current one; the second places them in directory "/a1/tf":

```
ar x /a1/tf/q.a
(cd /a1/tf; ar x q.a)
```

The *opt* command sets various flags in the Shell. For example, "opt -p prompt-str" changes the Shell's interactive prompt sequence from "% " to *prompt-str.*[4] Typing "opt -v" causes the Shell to enter verbose mode, in which it prints each command line before executing it {4.1}. Try this at the terminal to see how the Shell scans arguments. The output can be turned off by typing "opt +v".

The *login* command causes the Shell to execute the *login* program directly, permitting a new login without re-dialing. A related command is *su*, which permits you to act with someone else's access permissions *without* making you login again.

Wait causes the Shell to suspend execution until all of its child processes have terminated. It is used to assure termination of asynchronous processes.

When you login or use *su*, the Shell is invoked to read your commands, but if your current directory contains a file named ".profile", the Shell reads it *before* reading commands from your terminal; ".profile" often contains commands that set tab stops and terminal delays, read mail, etc. See ".profile" in {6}.

4. The default prompt string "% " is inconvenient for certain display (CRT) terminals.

4. USING THE SHELL AS A COMMAND: SHELL PROCEDURES

4.1 Invoking the Shell

The Shell is an ordinary command and may be invoked in the same way as other commands:

sh file [args]	A new instance of the Shell is explicitly invoked to read *file*. Arguments, if any, can be manipulated as described in {4.2}.
sh −v file [args]	This is equivalent to putting "opt −v" at the beginning of *file*. Each command line in *file* is printed before it is executed, thus tracing the progress of execution. *This is an important debugging aid.*
file [args]	If *file* is marked executable, and is neither a directory nor a load module, the effect is that of "sh file [args]", except that *file* may be found by the search procedure described in {3.7}.

4.2 Passing Arguments to the Shell

When a command line is scanned, any character sequence of the form $*n* is replaced by the *n*th argument to the Shell, counting the name of the file being read as $0. This notation permits direct reference to the file name and up to 9 arguments. Additional arguments can be processed using the *shift* command. It shifts arguments to the left; i.e., the value of $1 is thrown away, $2 replaces $1, $3 replaces $2, etc.; the rightmost argument becomes null. For example, consider the (executable) file "ripple" below. *Echo* writes its arguments to the standard output; *if, exit,* and *goto* are discussed later, but perform fairly obvious functions.[5] The form "$1" is used rather than '$1' because it is the value of the first argument that is desired, rather than the literal two-character string "$1":

```
: loop
if "$1" = "" exit
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
shift
goto loop
```

If the file were invoked by "ripple a b c", it would print:

```
a b c
b c
c
```

The "shift *n*" form of *shift* has no effect on the arguments to the left of the *n*th argument; the *n*th argument is discarded, and the higher-numbered ones shifted. Thus, "shift" is equivalent to "shift 1" (as is "shift 0").

The notation $* causes substitution of *all* current arguments except $0. Thus, the *echo* line in the "ripple" example above could be written in a better way as:

```
echo $*
```

These two *echo* commands are *not* equivalent: the first prints at most nine arguments; the second prints all its arguments. The $* notation is more concise and is less error-prone. One obvious application is in passing an arbitrary number of arguments to the *nroff* text formatter:

```
nroff −h −rT1 −T450 −mm $*
```

It is important to understand the sequence of actions used by the Shell in substituting arguments. First, the Shell reads one line of input, making all substitutions in a single pass; no rescanning is performed. Second, the Shell parses the resulting line. Third, the Shell executes all of the commands in that line. Thus, it is impossible for a command in a line to affect the argument values substituted into that same line. For example, the following sequence prints the same value twice, because the *shift* has no effect on the line in which it appears:

```
echo $1; shift; echo $1
```

5. Much better ways of coding this procedure are shown later. Lines that begin with ":" are labels and/or comments {4.5.1}.

On the other hand, the next sequence prints the first argument, followed by the second:

```
echo $1
shift
echo $1
```

4.3 Shell Variables

The Shell provides 26 string variables, $a through $z. Those in the range $a through $m are initialized to null strings at the beginning of execution and are never modified except by explicit user request. Some variables in the range $n through $z have specific initial values and may possibly be changed implicitly by the Shell during execution. A variable is assigned a value as follows:

```
= letter [ arg1 [ arg2 ] ]
```

If *arg1* is given, its value is assigned to the variable corresponding to *letter*. If two arguments are given, and if *arg1* is a null string, the value of *arg2* is assigned to the variable, permitting a convenient default mechanism. If neither *arg1* nor *arg2* are given, a single line is read from the standard input, and the resulting string (with the new-line character, if any, removed) is assigned to the variable.

The following are examples of simple assignments. You may omit quotes around the arguments if you are sure that they contain no special characters:

```
= a "$1"
= b '*****'
= c /usr/news/.mail
```

The procedure below illustrates the use of a default argument. If an argument is given, mail is read from it. Otherwise, mail is read from "/usr/news/.mail":

```
= a "$1" /usr/news/.mail
mail -f $a
```

The "=" command is often used to capture the output of a program. For example, *date* writes the current time and date to its standard output. The following line saves this value in $d:

```
date | = d
```

This works just as well in longer pipelines. The following saves in $a the number of logged-in users:

```
who | wc -l | = a
```

Another use is in the writing of *interactive* Shell procedures. The following example is part of a procedure to ask the user what kind of terminal is being used, so that tabs and delays can be set and other useful actions taken. The "</dev/tty" indicates a redirection of the standard input to the user terminal; it is *not* seen as an argument to "=", but rather causes the variable to be set to the next line typed by the user:

```
echo 'terminal?'
= a </dev/tty
```

Several variables are currently assigned special meanings:

$n records the number of arguments passed to the Shell, not counting the name of the Shell procedure itself. Thus, "sh file arg1 arg2 arg3" sets $n to 3. Its primary use is in checking for the required number of arguments:

```
if $n -lt 2 then
        echo 'two or more args required'; exit
endif
```

Shift never changes the value of $n.

$p permits alteration of the ordered list of directory pathnames used when searching for commands. It contains a sequence of directory names (separated by colons) that are to be used as search prefixes, ordered from left to right. The current directory is indicated by a null string.

By default, $p is initialized to a value producing the effect described in {3.7}: ":/bin:/usr/bin". A user could possess a personal directory of commands (say, /a1/tf/jtb/bin) and cause it to be searched *before* the other three directories by using:

```
= p /a1/tf/jtb/bin::/bin:/usr/bin
```

$r gives the value of the return code of the command most recently executed by the Shell. It is a string of digits; most commands return "0" to indicate successful completion. For example, the "=" command returns "0" if two arguments are given and the first is not null, or if a line is actually read from the input. When the Shell terminates, it returns the current value of $r as its own return code.

$s is initialized to the name of the user's *login directory*, i.e., the directory that becomes the current directory upon completion of a login (e.g., "/a1/tf/jtb"). Using this variable helps one to keep full pathnames out of Shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

$t is initialized to the user's terminal identification, a single letter or digit. The terminal can be manipulated using the file name "/dev/tty$t" or just "/dev/tty" alone. The latter is a generic name for the user's terminal.

$w is initialized to the first component of $s, i.e., it is the name of the file system (such as "/a1") in which the login directory is located. Like $s, it is used to avoid pathname dependencies, but is more useful than $s for projects involving many users.

$z is initialized to "/bin/sh". The command named by $z is the one that actually reads the Shell procedures invoked implicitly. The user can alter the choice of the Shell by overriding this value {4.4}. This facility is very useful when there are several different Shells in a system. This may occur because different groups of users want different Shells, or when a new Shell is being tested.

In addition to the above variables, the following read-only variable is provided:

$$ contains a 5-digit number that is the unique process number of the current Shell. Its most common use is in generating unique names for temporary files. Unlike many other systems, PWB/UNIX provides no separate mechanism for the automatic creation and deletion of temporary files: a file exists until it is explicitly removed. Temporary files are generally undesirable objects: the PWB/UNIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occur, especially for multi-user database applications. The following example of $$ usage also illustrates the helpful practice of creating temporary files in a directory used only for that purpose:

```
ls >$s/tmp/$$
... commands    (some of which use $s/tmp/$$)
: 'clean up at end'
rm $s/tmp/$$
```

4.4 Initialization of $p and $z by the .path File

The user may request automatic initialization of each Shell's $p (and $z) by creating a file named ".path" in the login directory. The first (or only) line should be of the form shown for $p {4.3}. If present, the second line should be the full pathname of a Shell. Every instance of the Shell looks for that ".path" file and initializes its own $p (and $z) from it, if ".path" exists. Otherwise, ":/bin:/usr/bin" and "/bin/sh" are the values used, respectively. Thus, the ".path" information is available to all of the user's Shells, but changing $p or $z in one Shell does *not* affect these variables in other Shells. In addition, ".path" is used in a consistent way by commands that must search for other commands, such as *nohup*, *nice*, and *time*.[6] This facility is heavily used in large projects, because it simplifies the sharing of procedures, and can be quickly altered to adapt to changes in organizational requirements.

6. If you plan to write such a command, investigate the *pexec* subroutine, which combines the search and execution code.

4.5 Control Structures

The Shell provides several commands that implement a variety of control structures. These commands are presented here in order of increasing complexity. See {6} for examples of these commands in the context of complete Shell procedures.

☛ *Several of the control commands must not be "hidden" on command lines (e.g., behind semi-colons ";"):*

else end endif endsw if switch while

Other control commands may be "hidden":

break breaksw continue exit goto next

4.5.1 Labels and Goto. The command ":" is recognized by the Shell, but is then treated as a null operation. One use of ":" is to define a label to act as a target for *goto*. Another use is to begin a comment line. However, it is a good idea to place comments in quotes {3.5} if they contain any characters that have a special meaning to the Shell, because the line *is* actually parsed, not just ignored. Using "goto label" causes the following actions:

- A *seek* is performed to move the read pointer to the beginning of the command file.
- The file is scanned from the beginning, searching for ": label", either alone on a line, or followed by a blank or tab.
- The read pointer is made to point at the line *after* the labeled line.

Thus, the only effect of *goto* is the adjustment of the Shell's file read pointer to cause the Shell to resume interpreting commands starting at the line following the labeled line. Invoking *goto* with an undefined label causes termination of the procedure {4.5.5}.

☛ *Avoid the "goto"—future versions of the Shell are not expected to allow it.*

4.5.2 If: Simple Conditional.

if conditional-expression command [args]

Whenever the conditional-expression is found to be *true, if* executes the command (via the *exec* system call), passing the arguments to it. Whenever the conditional-expression is *false, if* merely exits.

The following primaries can be used to construct the conditional-expression:

−r file	*true* if the named file exists and is readable by the user.
−w file	*true* if the named file exists and is writable by the user.
−s file	*true* if the named file exists and has a size greater than zero.
−d file	*true* if the named file is a directory.
−f file	*true* if the named file is an ordinary file.
s1 = s2	*true* if strings *s1* and *s2* are identical.
s1 != s2	*true* if strings *s1* and *s2* are *not* identical.
n1 −eq n2	*true* if the integers *n1* and *n2* are algebraically equal. Other algebraic comparisons are indicated by "−ne", "−gt", "−ge", "−lt", and "−le".
{ command }	the command is executed; a return code of 0 (yes, zero!) is considered *true*, any other value is considered *false*. Most commands return 0 to indicate successful completion.

These primaries may be combined with the following operators:

!	unary negation operator.
−a	binary logical *and* operator.
−o	binary logical *or* operator; it has lower precedence than "−a".
(expr)	parentheses for grouping. They must be escaped to remove their significance to the Shell. In the absence of parentheses, evaluation proceeds from left to right.

All of the operators, flags, and values are *separate* arguments to *if*, and must be separated by blanks. You must be careful to make sure that an argument actually appears and can be parsed correctly:

```
if "$1" = "" echo missing argument
if 0$1 = 0 echo missing argument
if 0"$1" = 0 echo missing argument
```

The first example guards against the possibility that $1 is omitted, null, or has embedded blanks; the second guards against the possibility that $1 has a value that causes parsing problems (such as "–r"), or that it is omitted or null; the third guards against all these problems. The following is dangerous:

```
if $1 = "" echo missing argument
```

because it would cause a syntax error in any of the above cases. Substitution of variables and arguments occurs effectively *before* parsing; thus, for example, if $1 were null, then after substitution the line would read:

```
if = "" echo missing argument
```

In this case, $1 without quotes yields no argument at all (on the other hand, "$1" would have yielded an argument whose value is the null string). It is generally desirable to quote arguments (with double quotes—see 3.5]), especially when they might possibly contain blanks or other characters that have a special meaning to the Shell. Examples of the use of *if* can be found in [6].

4.5.3 If—then—else—endif: Structured Conditional. A more general (and much more readable) form of *if* can be used:

```
if conditional-expression then
        ... commands
else
        ... commands
endif
```

The *else* and the commands following it may be omitted. It is legal to nest *if* commands, but there must be an *endif* to match every *then*.

When *if* is called with a command, using the form of [4.5.2], it acts as described there, deciding whether or not to execute the supplied command. When called with *then* instead of another command, *if* simply exits on a *true*, allowing the Shell to read and interpret the immediately following lines. On a *false*, *if* reads the file until it finds the next unmatched *else* or *endif*, thus skipping it and any other intervening lines. *Else* reads to the next unmatched *endif*. *Endif* is a null command.

These commands work together in a way that produces the appearance of a familiar control structure, although they do little but adjust the Shell's read pointer. Be warned that this implementation technique does *not* do a good job of diagnosing extra, missing, or hidden *if*, *else*, or *endif* commands [4.5]; if you suspect that there are such extra or missing commands, "opt –v" often helps [3.8,4.1].

4.5.4 Switch—breaksw—endsw: Multi-way Branch. The *switch* command manipulates the input file in a way quite similar to *if*. It is modeled on the "switch" statement of the C language [8], and like it, provides an efficient multi-way branch:

```
switch value
: label1
        ... commands
: label2
        ... commands
    .
    .
    .
: default
        ... commands
endsw
```

Switch reads the input until it finds:

- a statement label that matches *value*. The label may contain special characters as described in {3.4}; the method of matching is identical. A few of the many possible labels that could be used to match the value "thing.c" are:

```
thing.c   *.c   t*   *   ???????
```

- *default* used as a statement label (optional).
- the next unmatched *endsw* command.

Again, from the Shell's viewpoint, the only effect of *switch* is to adjust the read pointer so that the Shell effectively skips over part of the procedure, and then continues executing commands following the chosen label or *endsw*. For examples, see ".profile" and "fsplit" in {6}.

Value is obtained from an argument or from a variable; if the label *default* is present, it must be the last label in the list; it indicates a default action to be taken if *value* matches none of the preceding labels. The *switch* construct may be nested; labels enclosed by interior *switch-endsw* pairs are ignored during the execution of *switch*. *Breaksw* reads the input until the next unmatched *endsw* and is used to end the sequence of commands associated with a label. *Endsw* is a null command like *endif*.

4.5.5 End-of-file and Exit. When the Shell reaches the end-of-file, it terminates execution, returning to its parent the return code found in $r. The *exit* command simply seeks to the end-of-file and returns, setting the return code to the value of its argument, if any. Thus, a procedure can be terminated "normally" by using *exit 0*.

4.5.6 While—break—continue—end: Looping. A *while-end* pair delimits a loop. *Break* can be used to terminate execution of such a loop. *Continue* requests the execution of the next iteration of the loop:

```
while conditional-expression
        ... commands
end
```

While evaluates the conditional-expression, which is similar to that of *if* {4.5.2}. If the conditional-expression is *true, while* does nothing, permitting the following lines to be read and interpreted. If the conditional-expression is *false,* the input file is searched for a matching *end,* and command interpretation resumes with the next line. *While-end* groupings may be nested to a depth of three.

While treats a single, non-null argument as *true* and a single null argument or lack of arguments as *false.* This is convenient for the simple case that handles one argument per iteration:

```
while "$1"
        Do something with $1.
        shift
end
```

Break terminates execution of the smallest enclosing *while-end* group, causing execution to resume after the nearest following unmatched *end.* Exit from *n* levels is obtained by writing *n break* commands on the same line:

```
break; break; ...
```

Continue causes execution to resume at the preceding *while,* i.e., the one that begins the smallest loop containing the *continue.*

4.5.7 Conditional Operators || and &&. These operators enforce left-to-right execution of commands. In the line "cmd1 || cmd2", *cmd1* is executed and its return code examined. Only if it failed (exit code non-zero) is *cmd2* executed. It is thus a more terse notation for:

```
cmd1
if $r -ne 0 then
        cmd2
endif
```

The "&&" operator yields the inverse test: in "cmd1 && cmd2", the second command is executed only if the first succeeds (exit code zero). In the sequence below, each command is executed in order, until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

See "fsplit" and "writemail" in {6} for examples.

4.5.8 Next: Transfer to Another File. The command "next *name*" causes the Shell to abandon the current input and begin reading file *name*. *Next* with no arguments causes the Shell to read from the terminal. By creating a file that initializes Shell variables, then typing "next file" at the terminal, anyone can have a simple shorthand for setting a number of Shell variables with little typing. See "nx" in {6}.

4.6 Onintr: Interrupt Handling

As noted in {2.2}, a program may choose to "catch" an interrupt from the terminal, ignore it completely, or be terminated by it. Shell procedures can use *onintr* to obtain the same effects:

```
onintr [ label ]
```

Onintr takes several forms: "onintr label" yields the effect of "goto label" on receipt of an interrupt; "onintr" alone causes normal action to be restored, so that the process terminates on the next interrupt; "onintr –" causes interrupts to be ignored completely, not only by the Shell, but also by any commands invoked by it.

The most frequent use of *onintr* is to make sure that temporary files are removed at the end of a procedure. The example at the end of {4.3} typically would be written as:

```
onintr clean
ls >$s/tmp/$$
        ... commands
: clean
rm $s/tmp/$$
```

When "onintr label" is used, interrupts are effective at the time when the label is reached; it is often desirable to insert another *onintr* following the label. Even so, there may be a short "window" when the user can accidentally kill the procedure by causing repeated interrupts in quick succession.

4.7 Special I/O Redirections

As noted in {3.2}, when the Shell is invoked it expects to inherit from its parent an open standard input (file descriptor 0), standard output (file descriptor 1), and diagnostic output (file descriptor 2). Each of these is initially connected to the terminal.

4.7.1 Standard Input. When the Shell is invoked to read a command file, it saves the old standard input (in an invisible place), then opens the command file as the new standard input. The fact that commands inherit the *new* standard input is convenient for commands that read in-line data (editor scripts, etc.) not read by the Shell. However, this mechanism prevents a Shell procedure from acting as a filter or from reading the *old* standard input in the way that most C programs do. The Shell solves this problem by permitting the notation "<––" to allow a command to take its input from the old standard input, which the Shell has previously saved.[7]

Note that "</dev/tty" and "<––" usually have equivalent effects in a procedure invoked directly from the terminal. The effects differ in a procedure invoked from within another procedure, unless the first procedure takes care to invoke the second with "<––". In any case, "<––" is to be preferred because it can be used to read from a file or a pipe and is thus more general. See "fsplit" and "lower" in {6}.

---

7. The notation "––" arises from the concept of "standard input once removed", because many PWB/UNIX commands accept "–" in place of a file name to indicate that the *current* standard input should be read. This choice makes it impossible to redirect input from a file named "––". Fortunately, file names almost never begin with "–", because many commands expect "–" to signal a flag of some sort.

4.7.2 Standard Output. The use of ">/dev/tty" redirects output to the terminal, even if used in the middle of a pipeline. Shell procedures that act as filters sometimes need to do this. The redirection ">/dev/null" causes the standard output of a command to be thrown into a bottomless pit (presumably to feed the wumpus—see *wump*(VI)). This is used when you want to execute a command for its side-effects, but do not want to be bothered by its output.

4.7.3 Diagnostic Output. Most commands direct diagnostics to file descriptor 2 to make sure that they do not get lost down pipelines. Some situations require that this output go to some place other than the terminal. For example, a long-running procedure may be started, and then the terminal is hung up. In this case, it is helpful to save diagnostics in a file. A deficiency of the current Shell is the lack of syntax for redirecting the diagnostic output. The separate command *fd2* performs the required services:

fd2 [+] [−file] [−−file] command arguments ...

The "+" flag causes diagnostic output to be merged into the standard output. The second option writes that output to *file;* the third appends it to *file.* If the file name is omitted in the second or third cases, "msg.out" is used. If no flag is given, "−msg.out" is assumed.

4.8 Quoting Revisited

The main problem with quoting conventions is the need to treat "$" and "\" in ways flexible enough for convenient use with arguments and variables, but simple enough to be understandable, easy to implement, and unobtrusive in simple cases. In this respect, the current version of the Shell is far from elegant, but is reasonable in practice. The rules are:

- *Inside single* quotes, every character stands for itself without exception. A single quote is *not,* itself, allowed within single quotes.
- *Inside double* quotes, "\$" and "\"" stand for the characters "$" and """, respectively, but with all special meaning removed. All other characters, other than a pair of characters the first of which is an unescaped "$", behave exactly as they do within single quotes, including a "\" *not* followed by a "$" or a """.
- *Inside double* quotes and *outside* either kind of quotes, any two-character sequence whose first character is an unescaped "$" is replaced by the value of the corresponding Shell argument or variable; any variable that has no value (such as "$:") is replaced by a null string.
- *Outside* either kind of quotes, any two-character sequence whose first character is a "\" is replaced by the second character of that sequence, but with any special meaning removed.

4.9 Creation and Organization of Shell Procedures

A Shell procedure can be created in two simple steps. The first is that of building an ordinary text file. The second is that of changing the *mode* of the file to make it *executable,* thus permitting it to be invoked by "name args", rather than "sh name args". The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for longer-lived ones.

Here is the entire input needed to set up a simple procedure (the executable part of "draft" in {6}):

```
ed
a
nroff −rC3 −T450−12 −mm $*
.
w draft
q
chmod 755 draft
```

It may then be invoked as "draft file1 file2". If the Shell procedure "draft" were thus created in a directory whose name appears in the user's ".path" file, the user could change working directories and still invoke the "draft" command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the Shell to execute that file, then remove it. An alternate approach is that of using *next* to make the current Shell execute the new file, allowing use of existing Shell variables and avoiding the spawning of an additional process for another Shell. In some cases, the need for a temporary file may be eliminated by using the Shell in a pipeline.

Many users prefer to write Shell procedures instead of C programs. First, it is easy to create and maintain a Shell procedure because it is only an ordinary file of text. Second, it has no corresponding object program that must be generated and maintained. Third, it is easy to create a procedure "on the fly", use it a few times, then remove it. Finally, because Shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories of commands and/or Shell procedures are usually named "bin". Most groups of users sharing common interests have one or more "bin" directories set up to hold common procedures. Some users have ".path" files that list several such directories. Although you can have a number of such directories, it is unwise to go overboard—it may become difficult to keep track of your environment, and efficiency may suffer {7.3}.

5. MISCELLANEOUS SUPPORTING COMMANDS

Shell procedures can make use of almost any command. The commands described in this section are either used especially frequently in Shell procedures, or are explicitly designed for such use.

5.1 Echo: Simple Output

The *echo* command, invoked as "echo [args]", copies its arguments to the standard output, each followed by a single space, except the last argument, which is followed by a new-line; often, it is used to prompt the user for input, to issue diagnostics in Shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process (as in {3.4}) before issuing a command that does something drastic. The command "ls" is often replaced by "echo *" because the latter is faster and prints fewer lines of output.

Echo recognizes several escape sequences. A "\n" yields a new-line character. *Echo* normally appends a new-line character to its last argument; a "\c" is used to *suppress* that new-line character. The following prompts the user for input and allows input to be typed on the same line as the prompt:

```
echo 'enter name: \c'
= a </dev/tty
```

Echo also recognizes an octal escape sequence for any character, whether printable or not.

5.2 Pump: Shell Data Transfer

Pump is a filter that copies its standard input to its standard output with possible substitution of Shell arguments and variables:

```
pump [ -[ subchar ] ] [ + ] [ eofstr ]
```

Pump reads input until an end-of-file, or until it finds *eofstr* alone on a line. The default *eofstr* is "!". Normally, Shell arguments and variables are substituted in the data stream. The flag "-" suppresses all substitution, while the form "-subchar" causes *subchar* to be used as the indicator character for substitution of Shell variables and arguments, instead of "$". Escaping is handled as in strings enclosed by double quotes—the indicator character may be hidden by preceding it with "\". The "+" flag causes all leading tab characters in the input to *pump* to be eliminated; this permits that input to be indented for readability. A common use of *pump* is to get Shell variables into editor scripts—see "edfind" in {6}, for example. Because editor scripts may use "$" for other purposes, readability may be improved by using a *subchar* such as "%":

```
:       'in file $1, change every instance of $2 to $3'
:       'then delete all lines consisting only of $4'
if -r "$1" then
        pump -% + | ed $1
        g/%2/s//%3/g
        g/^%4$/d
        w
        !
else
        echo "$1: cannot open"
endif
```

Pump is often used to copy a few lines to another file:

```
pump >>logfile
here is $1
and here is $2 on a separate line
!
```

5.3 Expr: Expression Evaluation

Expr supports arithmetic and logical operations on integers, and PL/I-like "substr", "length", and "index" operators for string manipulation. It evaluates a single expression and writes the result to the standard output, typically piped into "=" to be assigned to a variable. Typical examples are:

```
:       'increment $a'
expr $a + 1 | = a

:       'put 3rd through last characters of $1 into $b'
:       'expr substr abcde 3 1000 returns cde (1000 is just a big number)'
expr substr "$1" 3 1000 | = b

:       'obtain length of $1'
expr length "$1" | = c
```

The most common uses of *expr* are in counting for loops and in using "substr" to pick apart strings.

5.4 Logname, Logdir, Logtty: Login Data

When a user logs in, he or she supplies a *login name* and a password. The *login* program searches the password file for that *login name* and obtains the name of the program to be executed by the user (normally the Shell), the directory to be made the current directory, and also a *userid,* a value ranging from 0 to 255. Most UNIX protection and identification mechanisms utilize the last item. Limiting the number of distinct users to 256 is no problem for most UNIX systems, but the original PWB/UNIX installation currently supports more than 1,000 users. However, it is not necessary to provide a distinct userid for every user. Project-oriented groups of users often choose to share one or two userids, in order to ease the problems caused by personnel absences, and also to ease the manipulation of shared files.[8] Although the members of such groups do not generally worry about being protected from each other, they need to be identified as distinct individuals by some programs, i.e., those that tag inter-user messages with user names or log the name of the user making a change to a source program. PWB/UNIX records the login name instead of discarding it after login. The *logname* command writes this name to the standard output, allowing it to be captured in a Shell variable. It can then be used to permit only selected users to execute a procedure, or can be included in logging information:

```
logname | = u
(echo "$u updated files on \c"; date) >>projectlog
```

The *logdir* and *logtty* commands are used in the same way as *logname;* they produce the same values as the initial values of $s and $t, respectively {4.3}.

---

8. Although some groups started by using one userid per person, it was discovered that these users often shared a single password. Thus, the possession of separate userids was considered more of a hindrance than a help.

6. EXAMPLES OF SHELL PROCEDURES

☛ *Some examples in this section may be quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name.*

.profile:

```
:       '.profile (automatically invoked on login) asks for terminal type,'
:       'reads a line from terminal, loops until a known type'
:       '(or empty line) is entered, sets terminal options appropriately,'
:       'asks for new directory name and changes to it, if one is given,'
:       'and then, if file nx exists, transfers to it'
while 1
        echo 'terminal:\c'
        = a </dev/tty
        switch "$a"
        :       'DASI450'
        : 450
                stty cr2; tabs +t450; break
        :       'GSI/DASI300'
        : gsi
        : 300
                stty cr2; tabs; break
        :       'HP264X'
        : hp
                stty cr0 nl0; tabs +thp; break
        :       'TI 700'
        : ti
                stty -tabs nl1 cr1; break
        : default
                if 0"$a" = 0 break
                echo "$a? try 450,gsi,hp,ti"
        endsw
end
echo "cd \c"
= b </dev/tty
if "$b" != "" then
        cd $b
endif
if -r nx then
        next nx
endif
```

Note: *Break* is used instead of *breaksw* in the above example to terminate the *while* loop, not just the *switch* construct.

copypairs:

```
:       'copypairs file1 file2 ...'
:       'copy file1 to file2, file3 to file4, ...'
while "$2"
        cp $1 $2
        shift; shift
end
if 0"$1" != 0 echo 'odd number of arguments'
```

Note: Remember that "shift; shift" is *not* the same as "shift 2". See next example for use of "shift 2".

copyto:

```
:       'copyto dir file ...'
:       'copy argument files to dir, making sure that at least'
:       'two arguments exist, that dir is a directory, and that'
:       'each additional argument is a readable file'
if $n -lt 2 then
        echo 'usage: copyto directory file ...'; exit
endif
if ! -d $1 then
        echo "$1 is not a directory"; exit
endif
while "$2"
        if ! -r $2 then
                echo "$2 not readable"
        else
                cp $2 $1
        endif
        shift 2
end
```

distinct1:

```
:       'distinct1'
:       'reads standard input, reports list of identifiers that'
:       'differ only in case, giving lower case form of each'
tr -cs '[A-Z][a-z][0-9]' '[\012*]' <-- | sort -u | tr '[A-Z]' '[a-z]' | sort | uniq -d
```

Note: This procedure is an example of the kind of process that is created by the "left-to-right" construction of a long pipeline. It may not be immediately obvious how this works. The *tr* translates all characters except letters and digits into new-line characters, and then "squeezes out" repeated new-line characters. This leaves each identifier (in this case, any contiguous sequence of letters and digits) on a separate line. *Sort* sorts the lines and emits only one line from any sequence of one or more repeated lines. The next *tr* converts everything to lower case, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The *uniq -d* prints once only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged; the two lines below are equivalent, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1 >tmp1; <tmp1 cmd2 >tmp2; <tmp2 cmd3; rm tmp[1-3]
```

Starting with a file of test data and working from left to right, each command is run taking its input from the previous file and putting its output in the next file. The final output file is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic "distinct1" with such a step-by-step process, using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc abc
```

Although pipelines can give a concise notation for complex processes, exercise some restraint lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This syndrome often yields incomprehensible code.

distinct2:

```
:       'distinct2'
:       'reads standard input, reports sorted list of identifiers that differ'
:       'in case only, listing all such distinct identifiers'
onintr cleanup
tr -cs '[A-Z][a-z][0-9]' '[\012*]' <-- | sort -u | tee t1$$ | tr '[A-Z]' '[a-z]' >t2$$
pr -s -t -l1 -m t1$$ t2$$ | sort +1 >t3$$
:       'third argument to pr in above line is "minus ell one"'
sort t3$$ >t4$$
uniq -u -1 t3$$ | sort | comm -23 t4$$ - | sort +1
: cleanup
rm t?$$
```

Note: This procedure is similar to the previous one, but provides more explicit information. As an exercise, work through this procedure in the way described above. The commands used here (plus *grep* and *sed*) form the basis for many "data stream" operations.

draft:

```
:       'draft file ...'
:       'prints the draft (-rC3) of a document on a DASI450 terminal in 12-pitch'
:       'using PWB/MM'
nroff -rC3 -T450-12 -mm $*
```

Note: Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags that cannot be given default values that are reasonable for all (or even most) users.

edfind:

```
:       'edfind file arg'
:       'find the last occurrence in file of a line that matches arg,'
:       'then print 3 lines (the one before, the line itself, and the one after)'
pump | ed - $1
?$2?;-,+p
!
```

Note: This illustrates the typical practice of using *pump* to substitute Shell variables into *ed* scripts.

edlast:

```
:       'edlast file'
:       'prints the last line of file, then deletes that line'
ed - $1
$p
$d
w
q
echo done
```

Note: This procedure illustrates the effects of a command that reads input from a file shared with the Shell.

fsplit:

```
:       'fsplit file1 file2'
:       'read standard input and split it into three parts:'
:       'append any line containing at least one letter to file1, any line'
:       'containing digits but no letters to file2, and throw the rest away'
= i 0; = j 0
while 1
        = a <-- || break
        expr $i + 1 | = i
        switch "$a"
        : *[A-Za-z]*
                echo "$a" >>$1; breaksw
        : *[0-9]*
                echo "$a" >>$2; breaksw
        : default
                expr $j + 1 | = j
        endsw
end
echo "$i lines read, $j thrown away"
```

Note: Each iteration of the loop reads a line from the input and analyzes it. The *break* terminates the loop only when "=" encounters an end-of-file.

☛ *Don't use the Shell to read a line at a time unless you must—it can be grotesquely slow* {7.2.1}.

loop:

```
:       'loop arg ...'
:       'one or more command lines'
:       'endloop'
:       'execute the group of command lines once for each argument,'
:       'substituting each argument as $1 in the command lines'
onintr cleanup
echo 'while "$1"' >tmp$$
pump - + endloop <-- >>tmp$$
echo 'shift \n end' >>tmp$$
next tmp$$; rm tmp$$
: cleanup
rm tmp$$
```

Note: Such a procedure is typically used from a terminal to repeat some commands for a list of arguments. It creates a temporary file that sandwiches user input between a *while* and *shift-end.* It then transfers to that file. For example, all files in the current directory could be copied to "place" by:

```
loop *
cp $1 place
echo $1 copied
endloop
```

lower:

```
:       'lower'
:       'reads standard input, converts it to lower case, writes to standard output'
:       'can thus be used in a pipeline if desired'
tr '[A-Z]' '[a-z]' <--
```

Note: This is the most common type of use for "<--".

mkfiles:

```
:       'mkfiles prefix [number]'
:       'makes number (default = 5) files, named prefix1, prefix2, ...'
= a "$2" 5
= i 1
while $i -le $a
        cp /dev/null $1$i
        expr $i + 1 | = i
end
```

null:

```
:       'null file ...'
:       'create each of the named files as an empty file'
while "$1"
        cp /dev/null $1
        shift
end
```

nx:

```
:       'next nx'
:       'asks for module name, initializes variables to useful values,'
:       'prints variables.  Note that variables are set within the invoking Shell,'
:       'so nx can be invoked only from terminal or from .profile'
= a /sys/source/s1
= b /usr/man/man1
echo "m: \c"
= m </dev/tty
= g "get -e s.$m; ed $m"
= d "delta s.$m"
pump
a: $a    b: $b
d: $d    g: $g    m: $m
!
next
```

phone:

```
:       'phone initials'
:       'prints the phone number(s) of person with given initials'
echo 'inits     ext     home'
grep "^$1"
abc     1234    999-2345
def     2234    583-2245
ghi     3342    988-1010
xyz     4567    555-1234
```

writemail:

```
:       'writemail message user'
:       'if that user is logged in, write message on terminal;'
:       'otherwise, mail it to that user'
echo "$1" | ( write "$2" || mail "$2" )
```

Note: Replacing "echo" above by "pump . <--" writes or mails the standard input, in the same way as the *mail* command.

7. EFFECTIVE AND EFFICIENT SHELL PROGRAMMING

7.1 Overall Approach

This section outlines strategies for writing "efficient" Shell procedures, i.e., ones that do not waste resources unreasonably in accomplishing their purposes. In the author's opinion, the primary reason for choosing the Shell procedure as the implementation method is to achieve a desired result at a minimum *human* cost. Emphasis should *always* be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, one should not worry about optimizing procedures unless they are intolerably slow or are known to consume a lot of resources.

The same kind of iteration cycle should be applied to Shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the *time* command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by, for instance, variations in system load.

7.2 Approximate Measures of Resource Consumption

7.2.1 Number of Processes Generated. When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of spawning processes. The CPU overhead per process lies in the range of 0.07 to 0.1 seconds, depending on the specific hardware configuration. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another Shell.

If you are worried about efficiency, it is important to know which commands are currently built into the Shell, and which are not. Here is the alphabetical list of those that are built-in:

:	chdir	endsw	newgrp	shift
=	continue	exit	next	switch
break	else	goto	onintr	test
breaksw	end	if	opt	wait
cd	endif	login	pump	while

Pump actually executes as a child process, i.e., the Shell does a *fork*, but no *exec;* "()" executes in the same way. Any command *not* in the above list requires both *fork* and *exec.*

The user should always have at least a vague idea of the number of processes generated. In the bulk of observed procedures, the number of processes spawned (not necessarily simultaneously) can be described by:

processes = k*n + c

where *k* and *c* are constants, and *n* is the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero. Any procedures whose complexity measures include n^2 terms or higher powers of *n* are likely to be intolerably expensive.

As an example, here is an analysis of procedure "fsplit" of [6]. For each iteration of the loop, there is one *expr* plus either an *echo* or another *expr*. One additional *echo* is executed at the end. If *n* is the number of lines of input, the number of processes is 2*n+1. On the other hand, the number of processes in the following (equivalent) procedure is 12, regardless of the number of lines of input:

fsplit2:

```
onintr cleanup
= b '[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]'
cat <-- >tmp$$
grep "$b" tmp$$ >tmp$$1
grep -v "$b" tmp$$ | grep "[0123456789]" >tmp$$2
cat tmp$$1 >>$1 ; cat tmp$$2 >>$2
wc -l <tmp$$ | = i
wc -l <tmp$$1 | = j
wc -l <tmp$$2 | = k
expr $i - $j - $k | = a
echo "$i read, $a thrown away"
:       cleanup
rm tmp$$*
```

This version is often ten times faster than "fsplit", and it is even better for larger input files.

Some types of procedures should *not* be written using the Shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C.

☛ *Shell procedures should not be used to scan or build files a character at a time.*

7.2.2 Number of Bytes of Data Accessed. It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the "shrinkers" first when the order is irrelevant. Which of the following is likely to be faster?

```
sort file | grep pattern
grep pattern file | sort
```

7.2.3 Directory Searches. Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of *cd* can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system).[9]

```
time sh -c 'ls -l /usr/bin/* >/dev/null'
time sh -c 'cd /usr/bin; ls -l * >/dev/null'
```

7.3 Efficient Organization

7.3.1 Directory Search Order and the .path File. The ".path" file is a popular and convenient mechanism for organizing and sharing procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead that occurs in a subtle, but avoidable way.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current $p variable. As an example, consider the effect of invoking *nroff* (/usr/bin/nroff) when $p is ":/bin:/usr/bin". The sequence of directories read is: ".", "/", "/bin", "/", "/usr", and "/usr/bin", i.e., a total of six directories. A long ".path" can increase this number significantly.

The vast majority of command executions are of commands found in "/bin" and, to a lesser extent, in "/usr/bin". Careless ".path" setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best (at least with regard to efficiency):

---

9. You may have to do some reading in the *PWB/UNIX User's Manual* [3] to understand exactly what is going on in these examples.

```
:/a1/tf/jtb/bin:/a1/tf/bin:/bin:/usr/bin
:/bin:/a1/tf/jtb/bin:/a1/tf/bin:/usr/bin
:/bin:/usr/bin:/a1/tf/jtb/bin:/a1/tf/bin
/bin::/usr/bin:/a1/tf/jtb/bin:/a1/tf/bin
```

The first one above should be avoided. The others are acceptable—choice among them is dictated by the rate of change in the set of commands kept in "/bin" and "/usr/bin".

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by changing $p to resemble the last of the above four examples.

7.3.2 Good Ways to Set up Directories. It is wise to avoid directories that are larger than necessary. You should be aware of several "magic sizes". A directory that contains entries for up to 30 files (plus the required "." and "..") fits in a single disk block and can be searched very efficiently. One that has up to 254 entries is still a "small" file; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most.

ACKNOWLEDGMENTS

The Shell was originally written by K. Thompson; its basic structure has remained unchanged since then, although many features (and some warts!) have been added. The PWB/UNIX extensions were added by R. C. Haight, A. L. Glasser, and the author. Some constructs have been derived from similar ones in the recent Shell written by S. R. Bourne. A number of colleagues provided helpful comments during the writing of this tutorial; T. A. Dolotta, in addition, provided a great deal of editorial assistance. Finally, many thanks must go to the PWB/UNIX user community, and especially M. H. Bianchi and J. T. Burgess, who provided many suggestions and examples.

REFERENCES

[1] Bianchi, M. H., and Wood, J. L. A User's Viewpoint on the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering,* pp. 193-99, Oct. 13-15, 1976.

[2] Dolotta, T. A., and Mashey, J. R. An Introduction to the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering,* pp. 164-68, Oct. 13-15, 1976.

[3] Dolotta, T. A., Haight, R. C., and Piskorik, E. M., eds. *PWB/UNIX User's Manual—Edition 1.0.* Bell Laboratories, May 1977.

[4] Kernighan, B. W., and Plauger, P. J. Software Tools. *Proc. First National Conference on Software Engineering,* pp. 8-13, Sept. 11-12, 1975.

[5] Kernighan, B. W., and Plauger, P. J. *Software Tools.* Reading, MA: Addison-Wesley, 1976.

[6] Mashey, J. R. Using a Command Language as a High-Level Programming Language. *Proc. Second Int. Conf. on Software Engineering,* pp. 169-76, Oct. 13-15, 1976.

[7] Mashey, J. R. PWB/UNIX Documentation Roadmap. Bell Laboratories, 1977.

[8] Ritchie, D. M. C Reference Manual. Bell Laboratories, 1977.

[9] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* 17(7):365-75, July 1974.

[10] Thompson, K. The UNIX Command Language. In *Structured Programming—Infotech State of the Art Report,* pp. 375-84. Infotech International Limited, Nicholson House, Maidenhead, Berkshire, England, 1976.

[11] Thompson, K., and Ritchie, D. M. *UNIX Programmer's Manual—Sixth Edition.* Bell Laboratories, May 1975.

UNIX For Beginners

Brian W. Kernighan

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help new users get started on UNIX. It covers:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-console communication, the file system, printing files, redirecting I/O, pipes, and the shell.

- document preparation — a brief tutorial on the ROFF formatter for beginners, hints on preparing documents, and capsule descriptions of some supporting software.

- UNIX programming — using the editor, programming the shell, programming in C, other languages.

There is also an annotated UNIX bibliography.

UNIX for Beginners

Brian W. Kernighan
Bell Laboratories, Murray Hill, N. J.

In many ways, UNIX is the state of the art in computer operating systems. From the user's point of view, it is easy to learn and use, and presents few of the usual impediments to getting the job done.

It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to point out high spots for new users, so they can get used to the main ideas of UNIX and start making good use of it quickly.

This paper is not an attempt to re-write the *UNIX Programmer's Manual;* often the discussion of something is simply "read section x in the manual." (This implies that you will need a copy of the *UNIX Programmer's Manual.)* Rather it suggests in what order to read the manual, and it collects together things that are stated only indirectly in the manual.

There are five sections:

1. Getting Started: How to log in to a UNIX, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which UNIX you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use UNIX effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX. This section contains advice, but not extensive instructions on any of the formatting programs.
4. Writing Programs: UNIX is an excellent vehicle for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages that UNIX provides.
5. A UNIX Reading List. An annotated bibliography of documents worth reading by new users.

I. GETTING STARTED

Logging In

Most of the details about logging in are in the manual section called "How to Get Started" (pages *iv-v* in the 5th Edition). Here are a couple of extra warnings.

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number. UNIX is capable of dealing with a variety of terminals: Terminet 300's; Execuport, TI and similar portables; video terminals; GSI's; and even the venerable Teletype in its various forms. But note: UNIX will not handle IBM 2741 terminals and their derivatives (e.g., some Anderson-Jacobsons, Novar). Furthermore, UNIX is strongly oriented towards devices with *lower case.* If your terminal produces only upper case (e.g., model 33 Teletype), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device: speed (if it's variable) to 30 characters per second, lower case, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal. UNIX should type "login:" at you. If it types garbage, you may be at the wrong speed; push the 'break' or 'interrupt' key once. If that fails to produce a login message, consult a guru.

When you get a "login:" message, type your login name *in lower case.* Follow it by a RETURN if the terminal has one. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it, again followed by a RETURN. (On M37 Teletypes always use NEWLINE or LINEFEED in place of RETURN).

The culmination of your login efforts is a percent sign "%". The percent sign means that UNIX is ready to accept commands from the terminal. (You may also get a message of the day just before the percent sign or a notification that you have mail.)

Typing Commands

Once you've seen the percent sign, you can type commands, which are requests that UNIX do something. Try typing

date

followed by RETURN. You should get back something like

Sun Sep 22 10:52:29 EDT 1974

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. We won't show the carriage returns, but they have to be there.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

pjp	ttyf	Sep 22 09:40
bwk	ttyg	Sep 22 09:48
mel	ttyh	Sep 22 09:58

The time is when the user logged in.

If you make a mistake typing the command name, UNIX will tell you. For example, if you type

whom

you will be told

whom: not found

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed. You can often fix this by logging out and logging back in. Or you can read the description of the command stty in section I of the manual. This will also tell you how to get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs. If it does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before the carriage return has been typed, there are two ways to recover. The sharp-character "#" erases the last character typed; in fact successive uses of "#" erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

dd#atte##e

is the same as "date".

The at-sign "@" erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an "@" and start over (on the same line!).

What if you must enter a sharp or at-sign as part of the text? If you precede either "#" or "@" by a backslash "\", it loses its erase meaning. This implies that to erase a backslash, you have to type two sharps or two at-signs. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Readahead

UNIX has full readahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away by UNIX and interpreted in the correct order. So you can type two commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). There are exceptions, like the text editor, where DEL stops whatever the program is doing but leaves you in that program. You can also just hang up the phone. The "interrupt" or "break" key found on most terminals has no effect.

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login name-of-new-user

and let someone else use the terminal you were on. It is not sufficient just to turn off the terminal. UNIX has no time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can send and receive letters from other users of the system. To read your mail, issue the command

mail

Your mail will be printed, and then you will be asked

Save?

If you do want to save the mail, type *y*, for "yes"; any other response means "no".

How do you send mail to someone else? Suppose it is to go to "joe" (assuming "joe" is someone's login name). The easiest way is this:

mail joe
now type in the text of the letter
on as many lines as you like ...
after the last line of the letter
type the character "control-d",
that is, hold down "control" and type
a letter "d".

And that's it. The "control-d" sequence, usually called "EOT", is used throughout UNIX to mark the end of input from a terminal, so you might as well get used to it.

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail** (I).

The notation **mail** (I) means the command **mail** in section (I) of the *UNIX Programmer's Manual.*

Writing to other users

At some point in your UNIX career, out of the blue will come a message like

Message from joe...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you're editing, you can escape temporarily from the editor — read the manual.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

Joe types "write smith" and waits.
Smith types "write joe" and waits.
Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing (o), which stands for "over".
Now Smith types a reply, also terminated by (o).
This cycle repeats until someone gets tired; he then signals his intent to quit with (o+o), for "over and out".
To terminate the conversation, each side must type a "control-d" character alone on a line. ("Delete" also works.) When the other person types his "control-d", you will get the message "EOT" on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type "control-d".

On-line Manual

The UNIX Programmer's Manual is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. It's also useful for getting the most up-to-date information on a command. To print a manual section, type "man section-name". Thus to read up on the **who** command, type

man who

If the section in question isn't in part I of the manual, you have to give the section number as well, as in

man 6 chess

Of course you're out of luck if you can't remember the section name.

II. DAY-TO-DAY USE

Creating Files — The Editor

If we have to type a paper or a letter or a program, how do we get the information stored in the machine? Most of these tasks are done with the UNIX "text editor" **ed**. Since **ed** is thoroughly documented in **ed** (I) and explained in *A Tutorial Introduction to the UNIX Text Editor,* we won't spend any time here describing how to use it. All we want it for right now is to make some *files.* (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file with some text in it, do the following:

```
ed      (invokes the text editor)
a       (command to "ed", to add text)
now type in
whatever text you want ...
.       (signals the end of adding text)
```

At this point we could do various editing operations on the text we typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, we write the information we have typed into a file with the editor command "w":

```
w junk
```

ed will respond with the number of characters it wrote into the file called "junk".

Suppose we now add a few more lines with "a", terminate them with ".", and write the whole thing out as "temp", using

```
w temp
```

We should now have two files, a smaller one called "junk" and a bigger one (bigger by the extra lines) called "temp". Type a "q" to quit the editor.

What files are out there?

The ls (for "list") command lists the names (not contents) of any of the files that UNIX knows about. If we type

```
ls
```

the response will be

```
junk
temp
```

which are indeed our two files. They are sorted into alphabetical order automatically, but other variations are possible. For example, if we add the optional argument "-t",

```
ls -t
```

lists them in the order in which they were last changed, most recent first. The "-l" option gives a "long" listing:

```
ls -l
```

will produce something like

```
-rw-rw-rw-  1 bwk   41 Sep 22 12:56 junk
-rw-rw-rw-  1 bwk   78 Sep 22 12:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (you got the same thing from ed). "bwk" is the owner of the file — the person who created it. The "-rw-rw-rw-" tells who has permission to read and write the file, in this case everyone.

Options can be combined: "ls -lt" would give the same thing, but sorted into time order. You can also name the files you're interested in, and ls will list the information about them only. More details can be found in ls (1).

It is generally true of UNIX programs that "flag" arguments like "-t" precede filename arguments.

Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,$p
```

ed will reply with the count of the characters in "junk" and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file ed can handle (about 65,000 characters or 4000 lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is cat, the simplest of all the printing programs. cat simply copies all the files in a list onto the terminal. So you can say

```
cat junk
```

or, to print two files,

```
cat junk temp
```

The two files are simply concatenated (hence the name "cat") onto the terminal.

pr produces formatted printouts of files. As with cat, pr prints all the files in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will list "junk" neatly, then skip to the top of a new page and list "temp" neatly.

pr will also produce multi-column output:

```
pr -3 junk
```

prints "junk" in 3-column format. You can use any reasonable number in place of "3" and **pr** will do its best.

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **roff**, **nroff**, and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under **opr** and **lpr**. Which to use depends on the hardware configuration of your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving a file a new name), like this:

```
mv junk precious
```

This means that what used to be "junk" is now "precious". If you do an **ls** command now, you will get

```
precious
temp
```

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

```
cp precious temp1
```

makes a duplicate copy of "precious" in "temp1".

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

```
rm temp temp1
```

will remove all of the files named. You will get a warning message if one of the named files wasn't there.

Filename, What's in a

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive. Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We already saw, for example, that in the **ls** command, "ls -t" meant to list in time order. So if you had a file whose name was "-t", you would have a tough time listing it by name. There are a number of other characters which have special meaning either to UNIX as a whole or to numerous commands. To avoid pitfalls, you would probably do well to use only letters, numbers and the period. (Don't use the period as the first character of a filename, for reasons too complicated to go into.)

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

```
chap1
chap2
etc...
```

Or, if each chapter were broken into several files, you might have

```
chap1.1
chap1.2
chap1.3
 ...
chap2.1
chap2.2
 ...
```

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

```
pr chap1.1 chap1.2 chap1.3 .....
```

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

```
pr chap*
```

The "*" means "anything at all", so this translates into "print all files whose names begin with 'chap' ", listed in alphabetical order. This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the "shell" sh(I)). Using that fact, you can see

how to list the files of the book:

```
ls chap*
```

produces

```
chap1.1
chap1.2
chap1.3
...
```

The "*" is not limited to the last position in a filename — it can be anywhere. Thus

```
rm *junk*
```

removes all files that contain "junk" as any part of their name. As a special case, "*" by itself matches every filename, so

```
pr *
```

prints all the files (alphabetical order), and

```
rm *
```

removes *all files*. (You had better be sure that's what you wanted to say!)

The "*" is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9 of the book. Then you can say

```
pr chap[12349]*
```

The "[...]" means to match any of the characters inside the brackets. You can also do this with

```
pr chap[1-49]*
```

"[a-z]" matches any character in the range *a* through *z*. There is also a "?" character, which matches any single character, so

```
pr ?
```

will print all files which have single-character names.

Of these niceties, "*" is probably the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of "*", "?", etc., enclose the entire argument in quotes (single or double), as in

```
ls "?"
```

What's in a Filename, Continued

When you first made that file called "junk", how did UNIX know that there wasn't another "junk" somewhere else, especially since the person in the next office is also reading this tutorial? The reason is that generally each user of UNIX has his own "directory", which contains only the files that belong to him. When you create a new file, unless you take special action, the new file is made in your own directory, and is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files that UNIX knows about are organized into a (usually big) tree, with your files located several branches up into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the right set of branches.

To begin, type

```
ls /
```

"/" is the name of the root of the tree (a convention used by UNIX). You will get a response something like this:

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that UNIX knows about. On most systems, "usr" is a directory that contains all the normal users of the system, like you. Now try

```
ls /usr
```

This should list a long series of names, among which is your own login name. Finally, try

```
ls /usr/your-name
```

You should get what you get from a plain

```
ls
```

Now try

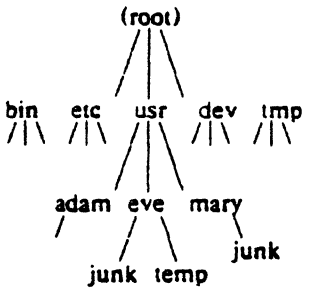
```
cat /usr/your-name/junk
```

(if "junk" is still around). The name

```
/usr/your-name/junk
```

is called the "pathname" of the file that you normally think of as "junk". "Pathname" has an obvious meaning: it represents the full name of the path you have to follow through the tree of directories to get to a particular file. It is a universal rule in UNIX that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:



Notice that Mary's "junk" is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

pr /usr/your-name/chap*

Similarly, you can find out what files your neighbor has by saying

ls /usr/neighbor-name

or make your own copy of one of his files by

cp /usr/your-neighbor/his-file yourfile

(If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be arranged. Each file and directory can have read-write-execute permissions for the owner, a group, and everyone else, to control access. See **ls** (I) and **chmod** (I) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.)

As a final experiment with pathnames, try

ls /bin /usr/bin

Do some of the names look familiar? When you run a program, by typing its name after a "%", the system simply looks for a file of that name. It looks first in your directory (where it typically doesn't find it), then in "/bin" and finally in "/usr/bin". There is nothing magic about commands like **cat** or **ls**, except that they have been collected into two places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

chdir /usr/your-friend

Now when you use a filename in something like **cat** or **pr**, it refers to the file in "your-friend's" directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact.

If you forget what directory you're in, type

pwd

("print working directory") to find out.

It is often convenient to arrange one's files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called book. So make one with

mkdir book

then go to it with

chdir book

then start typing chapters. The book is now found in (presumably)

/usr/your-name/book

To delete a directory, see **rmdir** (I).

You can go up one level in the tree of files by saying

chdir ..

".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX that the terminal can be replaced by a file for either or both of input and output. As one example, you could say

ls

to get a list of files. But you can also say

ls >filelist

to get a list of your files in the file "filelist". ("filelist" will be created if it doesn't already exist, or overwritten if it does.) The symbol ">" is used throughout UNIX to mean "put the output on the following file, rather than on the terminal". Nothing is produced on the terminal. As another example, you could concatenate several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 >temp
```

Similarly, the symbol "<" means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called "script". Then you can run the script on a file by saying

```
ed file <script
```

Pipes

One of the novel contributions of UNIX is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipe-line.

For example,

```
pr f g h
```

will print the files "f", "g" and "h", beginning each on a new page. Suppose you want them run together instead. You could say

```
cat f g h >temp
pr temp
rm temp
```

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

```
cat f g h | pr
```

The vertical bar means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr**, which formats it neatly.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines.

The Shell

We have already mentioned once or twice the mysterious "shell," which is in fact **sh** (I). The shell is the program that interprets what you type as commands and arguments. It also looks after translating "*", etc., into lists of filenames.

The shell has other capabilities too. For example, you can start two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a "%".

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don't want to wait around for the results before starting something else, you can say

```
ed file <script &
```

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately." Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to have said

```
ed file <script >lines &
```

which would save the output lines in a file called "lines".

When you initiate a command with "&", UNIX replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

```
kill process-number
```

You might also read **ps** (I).

You can say

```
(command-1; command-2; command-3) &
```

to start these commands in the background, or you can start a background pipeline with

```
command-1 | command-2 &
```

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell after all is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**; **date**; **who**) into a file, let's call it "xxx", and then run it with either

```
sh xxx
```

or

```
sh <xxx
```

This says to run the shell with the file "xxx" as input. The effect is as if you had typed the contents of "xxx" on the terminal. (If this is to be a regular thing, you can eliminate the need to type "sh"; see **chmod** (1) and **sh** (1).)

The shell has quite a few other capabilities as well, some of which we'll get to in the section on programming.

III. DOCUMENT PREPARATION

UNIX is extensively used for document preparation. There are three major *formatting* programs, that is, programs which produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. The simplest of these formatters is **roff**, which in fact is simple enough that if you type almost any text into a file and "roff" it, you will get plausibly formatted output. You can do better with a little knowledge, but basically it's easy to learn and use. We'll get back to **roff** shortly.

nroff is similar to **roff** but does much less for you automatically. It will do a great deal more, once you know how to use it.

Both **roff** and **nroff** are designed to produce output on terminals, line-printers, and the like. The third formatter, **troff** (pronounced "tee-roff"), instead drives a Graphic Systems phototypesetter, which produces very high quality output on photographic paper. This paper was printed on the phototypesetter by **troff**.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available which let you do things like paragraphs, running titles, multi-column output, and so on, with little effort. Regrettably, details vary from system to system.

ROFF

The basic idea of **roff** (and of **nroff** and **troff**, for that matter) is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page. In general, you don't have to spell out all of the possible formatting details. Most of them have "default values", which you will get if you say nothing at all. For example, unless you take special precautions, you'll get single-spaced output, 65-character lines, justified right margins, and 58 text lines per page when you **roff** a file. This is the reason that **roff** is so simple — most of the decisions have already been made for you.

Some things do have to be done, however. If you want a document broken into paragraphs, you have to tell **roff** where to add the extra blank lines. This is done with the ".sp" command:

```
this is the end of one paragraph.
.sp
This begins the next paragraph ...
```

In **roff** (and in **nroff** and **troff**), formatting commands consist of a period followed by two letters, and they must appear at the beginning of a line, all by themselves. The ".sp" command tells **roff** to finish printing any of the previous line that might be still unprinted, then print a blank line before continuing. You can have more space if you wish; ".sp 2" asks for 2 spaces, and so on.

If you simply want to ensure that subsequent text appears on a fresh output line, you can use the command ".br" (for "break") instead of ".sp".

Most of the other commonly-used **roff** commands are equally simple. For example you can center one or more lines with the ".ce" command.

```
.ce
Title of Paper
.sp 2
```

causes the title to be centered, then followed by two blank lines. As with ".sp", ".ce" can be followed by a number; in that case, that many input lines are centered.

".ul" underlines lines, and can also be followed by a number:

```
.ce 2
.ul 2
An Earth-shaking Paper
.sp
John O Scientist
```

will center and underline the two text lines. Notice that the ".sp" between them is not part of the line count.

You can get multiple-line spacing instead of the default single-spacing with the ".ls" command:

```
.ls 2
```

causes double spacing.

If you're typing things like tables, you will not want the automatic filling-up and justification of output lines that is done by default. You can turn this off with the command ".nf" (no-fill), and then back on again with ".fi" (fill). Thus

```
this section is filled by default.
.nf
here lines will appear just
as you typed them —
no extra spaces, no moving of words.
.fi
Now go back to filling up output lines.
```

You can change the line-length with ".ll", and the left margin (the indent) by ".in". These are often used together to make offset blocks of text:

```
.ll -10
.in +10
this text will be moved 10
spaces to the right and the
lines will also be shortened 10
characters from the right. The
"+" and "-" mean to change
the previous value by that
much. Now revert:
.ll +10
.in -10
```

Notice that ".ll +10" adds ten characters to the line length, while ".ll 10" makes the line ten characters *long*.

The ".ti" command indents (in either direction) just like ".in", except for only one line. Thus to make a new paragraph with a 10-character indent, you would say

```
.sp
.ti +10
New paragraph ...
```

You can put running titles on both top and bottom of each page, like this:

```
.he "left top"center top"right top"
.fo "left bottom"center bottom"right bottom"
```

The header or footer is divided into three parts, which are marked off by any character you like. (We used a double quote.) If there's nothing between the markers, that part of the title will be blank. If you use a percent sign anywhere in ".he" or ".fo", the current page number will be inserted. So to get centered page numbers with dashes around them, at the top, use

```
.he ""- % -""
```

You can skip to the top of a new page at any time with the ".bp" command; if ".bp" is followed by a number, that will be the new page number.

The foregoing is probably enough about **roff** for you to go off and format most everyday documents. Read **roff** (1) for more details.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

The second aspect of making change easy is not to commit yourself to formatting details too early. For example, if you decide that each paragraph is to have a space and an indent of 10 characters, you might type, before each,

```
.sp
.ti +10
```

But what happens when later you decide that it would have been better to have no space and an indent of only 5 characters? It's tedious indeed to go back and patch this up.

Fortunately, all of the formatters let you delay decisions until the actual moment of running. The secret is to define a new operation (called a *macro*), for each formatting operation you want to do, like making a new paragraph. You can say, in all three formatters,

```
.de PP
.sp
.ti +10
..
```

This *defines* ".PP" as a new **roff** (or **nroff** or **troff**) operation, whose meaning is exactly

```
.sp
.ti +10
```

(The ".." marks the end of the definition.) Whenever ".PP" is encountered in the text, it is as if you had typed the two lines of the definition in place of it.

The beauty of this scheme is that now, if you change your mind about what a paragraph should look like, you can change the formatted output merely by changing the definition of ".PP" and re-running the formatter.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of macros like ".PP", and then define them appropriately. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing and macro definitions. The packages of formatting commands that we mentioned earlier are simply collections of macros designed for particular formatting tasks.

One of the main differences between **roff** and the other formatters is that macros in **roff** can only be lines of text and formatting commands. In **nroff** and **troff**, macros may have arguments, so they can have different effects depending on how they are called (in exactly the same way that the ".sp" command has an argument, the number of spaces you want).

Miscellany

In addition to the basic formatters, UNIX provides a host of supporting programs. **eqn** and **neqn** let you integrate mathematics into the text of a document, in a language that closely resembles the way you would speak it aloud. **spell** and **typo** detect possible spelling mistakes in a document. **grep** looks for lines containing a particular text pattern (rather like the editor's context search does, but on a whole series of files). For example,

```
grep "ingS" chap*
```

will find all lines ending in the letters "ing" in the series of files "chap*". (It is almost always a good practice to put quotes around the pattern you're searching for, in case it contains characters that have a special meaning for the shell.)

wc counts the words and (optionally) lines in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

```
tr "[A-Z]" "[a-z]"
```

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand). **sort** sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing).

Most of these programs are either independently documented (like **eqn** and **neqn**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

IV. PROGRAMMING

UNIX is a marvelously pleasant and productive system for writing programs; productivity seems to be an order of magnitude higher than on other interactive systems.

There will be no attempt made to teach any of the programming languages available on UNIX, but a few words of advice are in order. First, UNIX is written in C, as is most of the applications code. If you are undertaking anything substantial, C is the only reasonable choice. More on that in a moment. But remember that there are quite a few programs already written, some of which have substantial power.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, say a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
 etc.
```

But instead you can do the job once and for all. Type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into "script". Now the command

```
ed <script
```

will produce the same output as the laborious hand typing.

The pipe mechanism lets you fabricate quite complicated operations out of spare parts already built. For example, the first draft of the **spell** program was (roughly)

```
cat ...     (collect the files)
| tr ...    (put each word on a new line,
             delete punctuation, etc.)
| sort      (into dictionary order)
| uniq      (strip out duplicates)
| comm      (list words found in text but
             not in dictionary)
```

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, and since UNIX already has a host of building-block programs, you can sometimes avoid writing a special purpose program merely by piecing together some of the building blocks with shell command files.

As an unlikely example, suppose you want to count the number of users on the machine every hour. You could type

```
date
who | wc -l
```

every hour, and write down the numbers, but that is rather primitive. The next step is probably to say

```
(date; who | wc -l) >>users
```

which uses ">>" to *append* to the end of the file "users". (We haven't mentioned ">>" before — it's another service of the shell.) Now all you have to do is to put a loop around this, and ensure that it's done every hour. Thus, place the following commands into a file, say "count":

```
: loop
(date; who | wc -l) >>users
sleep 3600
goto loop
```

The command : is followed by a space and a label, which you can then **goto**. Notice that it's quite legal to branch backwards. Now if you issue the command

```
sh count &
```

the users will be counted every hour, and you can go on with other things. (You will have to use **kill** to stop counting.)

If you would like "every hour" to be a parameter, you can arrange for that too:

```
: loop
(date; who | wc - l) >>users
sleep $1
goto loop
```

"$1" means the first argument when this procedure is invoked. If you say

```
sh count 60
```

it will count every minute. A shell program can have up to nine arguments, "$1" through "$9".

The other aspect of programming is conditional testing. The **if** command can test conditions and execute commands accordingly. As a simple example, suppose you want to add to your login sequence something to print your mail if you have some. Thus, knowing that mail is stored in a file called 'mailbox', you could say

```
if -r mailbox  mail
```

This says "if the file 'mailbox' is readable, execute the **mail** command."

As another example, you could arrange that the "count" procedure count every hour by default, but allow an optional argument to specify a different time. Simply replace the "sleep $1" line by

```
if $1x = x sleep 3600
if $1x != x sleep $1
```

The construction

```
if $1x = x
```

tests whether "$1", the first argument, was present or absent.

More complicated conditions can be tested: you can find out the status of an executed command, and you can combine conditions with 'and', 'or', 'not' and parentheses — see **if**(I). You should also read **shift**(I) which describes how to manipulate arguments to shell command files.

Programming in C

As we said, C is the language of choice: everything in UNIX is tuned to it. It is also a remarkably easy language to use once you get started. Sections II and III of the manual describe the system interfaces, that is, how you do I/O and similar functions.

You can write quite significant C programs with the level of I/O and system interface described in *Programming in C: A Tutorial*, if you use existing programs and pipes to help. For example, rather than learning how to open and close files you can (at least temporarily) write a program that reads from its standard input, and use **cat** to concatentate several files into it. This may not be adequate for the long run, but for the early stages it's just right.

There are a number of supporting programs that go with C. The C debugger, **cdb**, is marginally useful for digging through the dead

bodies of C programs. **db**, the assembly language debugger, is actually more useful most of the time, but you have to know more about the machine and system to use it well. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

You can instrument C programs and thus find out where they spend their time and what parts are worth optimising. Compile the routines with the "-p" option; after the test run use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but it's not super accurate or reproducible.

C programs that don't depend too much on special features of UNIX can be moved to the Honeywell 6070 and IBM 370 systems with modest effort. Read *The GCOS C Library* by M. E. Lesk and B. A. Barres for details.

Miscellany

If you *have* to use Fortran, you might consider **ratfor**, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **db**, **prof**, etc., are all virtually useless with Fortran programs.

If you want to use assembly language (all heavens forfend!), try the implementation language LIL, which gives you many of the advantages of a high-level language, like decent control flow structures, but still lets you get close to the machine if you really want to.

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly.

V. UNIX READING LIST

General:

UNIX Programmer's Manual (Ken Thompson, Dennis Ritchie, and a cast of thousands). Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only read section I.

The UNIX Time-sharing System (Ken Thompson, Dennis Ritchie). CACM, July 1974. An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

Document Preparation:

A Tutorial Introduction to the UNIX Text Editor. (Brian Kernighan). Bell Laboratories internal memorandum. Weak on the more esoteric uses of the editor, but still probably the easiest way to learn **ed**.

Typing Documents on UNIX. (Mike Lesk). Bell Laboratories internal memorandum. A macro package to isolate the novice from the vagaries of the formatting programs. If this specific package isn't available on your system, something similar probably is. This one works with both **nroff** and **troff**.

Programming:

Programming in C: A Tutorial (Brian Kernighan). Bell Laboratories internal memorandum. The easiest way to start learning C, but it's no help at all with the interface to the system beyond the simplest IO. Should be read in conjunction with

C Reference Manual (Dennis Ritchie). Bell Laboratories internal memorandum. An excellent reference, but a bit heavy going for the beginner, especially one who has never used a language like C.

Others:

D. M. Ritchie, UNIX Assembler Reference Manual.

B. W. Kernighan and L. L. Cherry, A System for Typesetting Mathematics, Computing Science Tech. Rep. 17.

M. E. Lesk and B. A. Barres, The GCOS C Library. Bell Laboratories internal memorandum.

K. Thompson and D. M. Ritchie, Setting Up UNIX.

M. D. McIlroy, UNIX Summary.

D. M. Ritchie, The UNIX I/O System.

A. D. Hall, The M6 Macro Processor, Computing Science Tech. Rep. 2.

J. F. Ossanna, NROFF User's Manual — Second Edition, Bell Laboratories internal memorandum.

D. M. Ritchie and K. Thompson, Regenerating System Software.

B. W. Kernighan, Ratfor—A Rational Fortran, Bell Laboratories internal memorandum.

M. D. McIlroy, Synthetic English Speech by Rule, Computing Science Tech. Rep. 14.

M. D. McIlroy, A Manual for the TMG Compiler-writing Language. Bell Laboratories

internal memorandum.

J. F. Ossanna, TROFF Users' Manual, Bell Laboratories internal memorandum.

B. W. Kernighan, TROFF Made Trivial, Bell Laboratories internal memorandum.

R. H. Morris and L. L. Cherry, Computer Detection of Typographical Errors, Computing Science Tech. Rep. 18.

S. C. Johnson, YACC (Yet Another Compiler-Compiler), Bell Laboratories internal memorandum.

P. J. Plauger, Programming in LIL: A Tutorial, Bell Laboratories internal memorandum.

Index

UNIX Programming

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on UNIX. The emphasis is on how to write programs that interface to the operating system. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the portable C I/O library; file system access
- standard UNIX file I/O
- low-level I/O
- executing commands from programs
- signals — interrupts, etc.

UNIX Programming

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories,
Murray Hill, New Jersey 07974

1. Introduction

This paper describes how to write UNIX programs that interface with the operating system in a non-trivial way. This includes programs that use files by name, that do large amounts of input or output, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1]. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you will have to be able to read the language roughly up to the level of *Programming in C — A Tutorial* [2]. You should also be familiar with UNIX itself to the level of *UNIX for Beginners* [3].

2. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the main program as an argument count **argc** and an array of character strings **argv** containing the arguments. By convention, **argv[0]** is the command name itself, so **argc** is always greater than 0.

Here is a program that simply echoes its arguments back to the terminal. (This is much like the command **echo**.)

```
main( argc, argv )
int argc;
char *argv[ ];
{
        int i;
        for( i=1; i < argc; i++ )
                printf( "%s ", argv[i] );
        putchar( '\n' );
}
```

main is called with two arguments, the argument count and the array of arguments. **argv** is a pointer to an array, whose individual elements are pointers to arrays of characters; each is terminated by '\0', so they can be treated as strings. **argv[0]** is the name of the command itself, so we start by printing **argv[1]** and loop until we've printed them all. Each **argv[i]** is a character array, so we use a '%s' in the **printf**.

A common convention in UNIX programs is that an argument which begins with '–' indicates a flag or option of some sort. For example, suppose we want a program to be callable as

```
prog −abc arg1 arg2 ...
```

where the '−' argument is optional; if it is present, it must be first and may be followed by any combination of the options 'a', 'b', and 'c'.

```
main(argc, argv)
int argc;
char *argv[ ];
{
        ...
        aflag = bflag = cflag  = 0;
        while( argc > 1 && argv[1][0] == '−' ) {
                for( i=1; (c=argv[1][i]) != '\0'; i++ )
                        if( c=='a' )
                                aflag++;
                        else if( c=='b' )
                                bflag++;
                        else if( c=='c' )
                                cflag++;
                        else
                                printf( "%c?\n", c );
                −−argc;
                ++argv;
        }
        ...
```

The statements

```
−−argc;
++argv;
```

drop the first argument from the list and adjust the count, so after interpreting the flag argument, the rest of the program is independent of whether or not it existed. This works because argv is a pointer which can be incremented. Notice also that, for greatest generality, the while and for loops in combination allow the user to write the options either as

```
program −a −b −c
```

or as

```
program −abc
```

This degree of generality is unfortunately not very common; most commands require one form or the other.

The argument count and the arguments are parameters to **main**. If you want to keep the arguments around so other routines can get at them, you will probably want to copy them to external variables.

3. Rudimentary Input and Output

The next several sections will discuss various aspects of input/output, including how to create, open, and close files from programs. There are several ways to do most of these operations; these sections are organized so easy things are described first.

3.1. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function **getchar()** returns the next input character each time it is called. Of course, a file may be substituted for the terminal by using the '<' convention: if **prog** uses **getchar**, then the command line

```
prog <file
```

causes **prog** to read **file** instead of the terminal. **prog** itself knows nothing about where its input is coming from. This is also true if the input comes from another program via the UNIX pipe mechanism:

```
otherprog | prog
```

will provide the input for **prog** from the output of **otherprog**.

getchar returns the value zero (often written as the null-character '\0') when it encounters the end of file (or an error) on whatever you are reading. Bear in mind that '\0' may be a legitimate value in some contexts. If it is, you can't use **getchar**; read ahead to the discussion of **getc**.

In a similar manner, **putchar(c)** puts the character **c** on the "standard output," which is also by default the terminal. The output can be captured on a file by using '>': if **prog** uses **putchar**,

```
prog >outfile
```

will write the output onto **outfile** instead of the terminal. And a pipe can be used:

```
prog | otherprog
```

puts the output of **prog** into the input of **otherprog**.

The function **printf**, which formats output in various ways, uses **putchar** to finally print the output, so output produced by **printf** also finds its way to the standard output.

A surprising number of programs read only one input and write one output; for such programs I/O with **getchar**, **putchar**, and **printf** may be entirely adequate, and it is almost always enough to get started. This is particularly true, given the UNIX pipe facility for connecting the output of one program to the input of the next. For example, here is a complete program that acts as a "filter" to strip out all ascii control characters from its input (except for newline and tab).

```
main( ) {
        int c;
        while( c=getchar( ) )
                if( (c>=' ' && c<0177) || c=='\t' || c=='\n' )
                        putchar(c);
        exit( 0 );
}
```

If it is necessary to treat multiple files, you can use **cat** to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to **exit** at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 7.3 discusses status returns in more detail.

3.2. File Descriptors

Before we go much further into our description of I/O, we have to talk about file descriptors. Any program which does any input or output does so by reading or writing files. This is true even though the file in question may actually be a device like the user's terminal. Associated with each file being used for input or output is a small non-negative integer called a "file descriptor;" whenever I/O is to be done on the file, the file descriptor is used to identify the file, instead of the name. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.)

In the most general case, to do I/O on a file you have to

Open the file for reading and/or writing — UNIX connects the name of the file with a file descriptor which it generates and returns to you. An alternative form of *open* will create the file if it doesn't exist.

Read or write on the file.

Close the file, which breaks the connection between name and descriptor.

And finally, you may want to

Buffer input and output if necessary for efficiency.

In the simplest cases, like **getchar** and **putchar**, all opening and closing is done for you; you only have to worry about reading and writing the right things. In more complicated situations, you have to do more work, but you gain flexibility.

getchar and **putchar** depend on the fact that when the command interpreter (the shell) runs a program, it opens three files, with file descriptors 0, 1, and 2. All of these are normally connected to the terminal, so if you read 0 and write 1 or 2, I/O is done on the terminal. If you use '<' or '>', the shell changes the default assignment for file 0 or 1 from your terminal to the named file, and opens it for you. This way your program need not know where its input comes from nor where its output goes, so long as it uses files 0 and 1. Naturally, **getchar** reads 0 and **putchar** writes 1. When the program terminates, the files are closed automatically. (We'll get back to file 2 in Section 3.4.)

3.3. Buffering the Standard Input and Output

If you are producing large amounts of output, you may find that programs which use **putchar** are slow, because each call to it actually requires a system call. You can speed up such a program markedly by buffering the output. (Generally UNIX buffers input automatically, which is almost always what is wanted. See Section 6 for how to turn off buffering.)

Buffering requires a little witchcraft. First we need a buffer area. This can be provided by referring to an external variable **fout** which is used by **putchar** for output. **fout** is actually a structure defined as part of **putchar**, but its internal structure needn't concern us now. Second, we have to connect the standard output to **fout**. Lastly, when all processing is done, any output remaining in the output buffer has to be flushed out; this is not done automatically. Putting all this together gives

```
main( ) {
        extern int fout;
        ...
        fout = dup( 1 );        /* buffer standard output */
        [ processing ... ]
        flush( );/* force out last buffer */
        exit( 0 );
}
```

This should probably be treated as black magic for now. Briefly, **putchar** buffers only if the file has a descriptor greater than 2. The call to **dup** creates a new file descriptor that refers to the same file as the original 1, but is guaranteed to be buffered. **flush** forces out everything that has collected in the buffer.

As an aside, there is an external variable **fin** used by **getchar** much as **putchar** uses **fout.**

3.4. Diagnostic Output — The Error File

If we are going to run our control-character stripping program, it might be nice to know whether it actually removed any characters. But at the same time we can't just print a message, because that will be sent to the same place as the data itself. That is, if we say

```
ccstrip <infile >outfile
```

we don't want **outfile** to contain a line saying "there were 3 bad characters."

Here is how to get this kind of diagnostic information separated from the standard output and placed on the terminal regardless. Just as the file descriptors 0 and 1 are predefined, so is file descriptor 2. Unless you go out of your way to change it, output written on file 2 will find its way to your terminal. Thus in simple cases, you can simply set **fout** to 2 to direct output to the terminal.

```
main( ) {
        extern int fout;
        ...
        flush( );/* clear out standard output */
        fout = 2;
        printf( "%d bad characters\n", badchar );
        exit( 0 );
}
```

This can get clumsy if the diagnostic output is to show up during the running of the program instead of all at the end, because you have to flush output for one file before switching to another, *each* time you switch.

4. The Portable C Library

The portable C library [3] was written by Mike Lesk to provide a set of high-level I/O routines that could be implemented on any machine with a C compiler, and thus permit some degree of program transferability. C programs which use this library for I/O can be moved, with essentially no change, between UNIX, GCOS, and IBM-TSO. At the same time, many of the details of buffering, file access, etc., are hidden from the user. The routines are somewhat bigger and slower than the analogous routines that are a standard part of UNIX, but they do handle a number of things automatically which other packages do not.

If you use programs from the portable C library, you have to ask specifically that it be searched when you compile or link-edit your program by writing '−lp" at the end of the arguments to the cc command:

```
cc prog.c −lp
```

4.1. File Opening and Closing

Before a file can be read or written, it has to be opened with the routine **copen**. **copen** has two arguments, the file name, and the type of access wanted (read, write, or append). Thus:

```
fd = copen( "/usr/bwk/foo", 'r' );
```

opens /usr/bwk/foo for reading. The value returned by **copen** is the file descriptor assigned by UNIX, to be used later for reading the file. If this number is −1, an error has occurred, so some defensive action has to take place. The usual code is

```
if( (fd=copen(name,mode)) == -1 )
        error( "Can't open file", name );
```

where **error** is some message printer.

mode is one of 'r', 'w', or 'a'. The arguments 'w' and 'a' mean writing and appending. If the file is to be opened for writing, and if it doesn't already exist, it will be created for you. If the argument is 'w' and if the file already exists, it will be truncated to zero length. If the argument is 'a', you will write at the end of the file in either case. (And of course if any of this fails, you will get a −1 error return.)

When you finish processing a file, you should close it explicitly with

```
cclose(fd);
```

where fd is the file descriptor handed you by **copen**. **cclose** will flush out any buffer contents remaining before closing the file; there is no **flush** command in the portable C library. Finally, when your program is done, you should call **cexit** which will close any open files and flush out their buffers; like **exit**, it then terminates the program and delivers its argument as termination status. By the way, there is a limit of 15 simultaneously open files, so if your program deals with many files it will have to call **cclose** explicitly.

Let us illustrate these routines with a program modeled after **grep**. **grep**, the general purpose pattern finder, has several arguments, some of which are file names. If there are no file names, it reads the standard input. And it writes on the standard output. That is,

```
grep pattern [optional list of input files]
```

will print all input lines that contain "pattern". Thus we write

```
main( argc, argv )
int argc;
char *argv[ ];
{
        char line[1000], *pattern;
        int i, fd;
        if( argc < 2 ) {
                printf( 2, "Usage: grep pattern [file...]\n" );
                cexit( 1 );
        }
        pattern = argv[1];
        i = 2;
        do {
                if( argc == 2 )
                        fd = 0;/* use standard input */
                else if( (fd=copen(argv[i], 'r')) == -1 ){
                        printf( 2, "can't open %s\n", argv[i] );
                        cexit( 1 );
                }
```

```
                while( gets( line, fd ) )
                        if( match( line, pattern ) )
                                puts( line );
                if( fd != 0 )
                        cclose( fd );
        } while( ++i < argc );
        cexit( 0 );
}
```

First the arguments are validated. Then the files are gone through in order; each is opened (if possible), and scanned in the **while** loop.

gets and **puts** are routines in the portable C library that read and write a line at a time. **match** is an unspecified routine (you write it!) that tells whether the line contains the pattern.

Notice that we have written a call to **printf** with a first argument of 2. If the first argument of **printf** is a small positive integer, this is assumed to be a file descriptor, and the output is sent there. (Caveat: this is true in the portable library only, unfortunately.) Thus any error messages go to the user's terminal.

There are a couple of other things to note in passing. First, the basic design of the program is that it allows input to be either from a set of files or from the standard input, and it writes on the standard output. This way the program can be used stand-alone or as part of a pipeline. It is important to design and implement programs this way whenever possible. Second, the program signals errors in two ways. The diagnostic output goes out onto the error file so it finds its way to your terminal instead of disappearing down a pipeline or into a file. Also the program returns a value when it calls [c]exit, so the success or failure of the command can be tested from within another program that uses this one as a sub-process.

4.2. Character I/O

The portable library provides two routines for I/O of individual characters. **cgetc** and **cputc** are quite analogous to **getchar** and **putchar**, except that they require an additional argument to specify a file descriptor. Thus

```
cgetc( 0 );  cputc( c, 1 );
```

reads the standard input and writes the standard output, while

```
cgetc( fd );  cputc( c, fd );
```

reads and writes **fd**. For convenience, **getchar** and **putchar** are also provided; they just call **cgetc** or **cputc** with the appropriate file descriptor argument. As an exercise you might write your own versions of **gets** and **puts** using **cgetc** and **cputc**.

4.3. Miscellaneous

The portable C library contains several other goodies, the most useful of which is probably the function **scanf**, which provides input format conversion similar to **printf** on output; it will convert strings to integers, floating point numbers, and so on. There is also a way to use **printf** to do in-core format conversion:

```
printf( -1, s, format, ... );
```

will put its output in string **s** instead of a file. The **ungetc** function can be used tp push characters back onto the input stream for re-reading. And the **ceof** function can be used to test explicitly for end-of-file, so data which contains null characters can be handled with **cgetc**.

5. Standard UNIX I/O

This section describes the I/O routines provided as part of "standard" UNIX. They do somewhat less for you than the portable library, but are more efficient, and perhaps more widely available. The essential difference is that the user has to supply the buffer for each file explicitly, and provide his own flushing of output before closing a file. These routines are described in sections II and III of the *UNIX Programmer's Manual* [1].

Let us illustrate by writing a simplified version of cmp, a program that compares two files byte by byte.

```
main( argc, argv )
int argc;
char *argv[ ];
{
        int c1, c2, byte;
        int buf1[259], buf2[259];
        if( argc != 3 )
                error( "Usage: cmp file1 file2" );
        if( fopen( argv[1], buf1 ) < 0 )
                error( "can't open %s", argv[1] );
        if( fopen( argv[2], buf2 ) < 0 )
                error( "can't open %s", argv[2] );
        for( byte=0 ; ; byte++ ) {
                c1 = getc( buf1 );
                c2 = getc( buf2 );
                if( c1 < 0 || c2 < 0 )
                        break;
                if( c1 != c2 )
                        printf( "%6u %3o %3o\n", byte, c1, c2 );
        }
        if( c1 == c2 )
                exit( 0 );
        if( c1 < 0 )
                printf( "eof 1\n" );
        else
                printf( "eof 2\n" );
        exit( 1 );
}

error( s1, s2 )
char *s1;
int s2;
{
        printf( s1, s2 );
        printf( "\n" );
        exit( 2 );
}
```

Files are opened with fopen, whose two arguments are the filename and a buffer, which is always declared int[259]. The buffer is actually used as a structure by the I/O routines (for example, the file descriptor goes into buf[0]), but we need not be concerned with that here; it is sufficient to note that the buffer is the connection between all the routines concerned with a file. As with copen, fopen returns a −1 if the access failed for any reason. error is a simple routine to print out a message and exit with the appropriate status return.

getc reads the input using the buffer argument to tell it what file to read. On end of file, **getc** returns the value -1, not zero, so it can be used to read data containing explicit zero characters, where **getchar** and **cgetc** are not suitable.

The situation for output is slightly more complicated. First, the output file may not exist. The routine **fcreat** allows for this:

```
fcreat( name, buf )
```

creates the file if it doesn't exist; if it does exist, it is truncated to zero length. To write on the file,

```
putc( ch, buf )
```

puts the character **ch** onto the file. When writing is done, call

```
fflush( buf )
```

to force out any left-over output, then call

```
close( buf[0] )
```

to close the file. Termination of the program, by calling **exit** or otherwise, closes all open files, but *it does not flush the output buffers.*

As an aside, the way to delete a file is simply to call

```
unlink( filename );
```

This returns −1 if the unlinking failed. Renaming a file uses both **link** and **unlink**:

```
if ( link( oldname, newname ) >= 0)
        unlink( oldname );
```

renames the file from "oldname" to "newname," taking care not to delete the file if the link failed, as it would, for example, if "newname" already existed.

6. Low-Level I/O

The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

6.1. I/O Proper

For the low-level I/O routines, files are identified directly by their file descriptors. Here is a simple version of **cp**, a program which copies one file to another.

```
main( argc, argv )
int argc;
char *argv[ ];
{
        int f1, f2, n;
        char buf[512];
        if( argc != 3 )
                error( "Usage: cp from to" );
        if( (f1=open(argv[1], 0) < 0 )
                error( "can't open %s", argv[1] );
        if( (f2=creat(argv[2], 0666)) < 0 )
                error( "can't open %s", argv[2] );
        while( (n=read(f1, buf, 512)) > 0 )
                write( f2, buf, n );
```

```
        exit( 0 );
}
```

open is rather like fopen except that instead of a buffer, its second argument tells what sort of access is required. 0 implies reading, 1 implies writing, and 2 is both read and write. A −1 is returned if any error takes place, for example if the file doesn't exist.

creat is similar to fcreat, except that the second argument is the protection mode in which the output file is to be created. "0666" is read and write permission for everyone. creat opens the file for writing only.

I/O is done by read and write. In both cases, the first argument is the file descriptor returned by a previous open or creat. The second argument is the place where the data is to come from or go to. The third argument is the number of bytes to be transferred. On reading, the actual number of bytes returned may be less than this value. Zero bytes implies end of file, a −1 implies an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The routine close may be used to close files after you are done with them. Termination of the program via exit or return from the main program closes all files, but since only 15 files can be open at once, close has to be used when many files are read or written.

It is instructive to see how read and write can be used to construct special versions of things like getchar, putchar, etc. For example, if you don't want buffered input, you have to write your own getchar:

```
getchar( ) {
        int c, n;
        c = 0;
        n = read( 0, &c, 1 );
        return( n>0 ? c : 0 );
}
```

At this point you might wonder why unbuffered input should be useful. In practice, it usually doesn't matter, but consider the following sequence of commands:

```
ed stuff
1,$ s/  */ /g
w stuff
q
opr stuff
```

which uses the editor to replace all sequences of multiple blanks in "stuff" by a single blank, then prints "stuff" off-line. Now imagine this pair of commands in a shell command file, and finally consider what would happen if the editor read its standard input with buffering. Since the command file contains commands to both the editor and the shell, the editor, if it read with a large enough buffer, would consume not only not only its own commands but also the following *opr* command which is intended for the shell. One can imagine complicated schemes for pushing back unwanted input, but the simplest approach for the editor, and analogous programs, is to read a single character at a time so they can't disturb input not intended for them. (Incidentally, the portable library's ungetc function doesn't solve this problem, since it just pushes input back into an internal buffer which vanishes along with its caller on termination.)

Similarly you can use the fact that printf calls putchar explicitly to make your own error-printing routines. If you provide a putchar like this

```
putchar(c) {
        write(2, &c, 1);
        return(c);
}
```

then calls to printf will write on the terminal, unbuffered.

As another example, this code duplicates the putchar and flush routines described above except that it always buffers:

```
char    buf[512];
int     fdes    1;
int     nleft   512;
char    *nextfree       &buf[0];

putchar( c )
{
        *nextfree++ = c;
        if ( --nleft <= 0 )
                flush( );
}

flush( )
{
        write( fdes, buf, nextfree - buf );
        nleft = 512;
        nextfree = buf;
}
```

6.2. Random Access

The seek routine provides a way to move around in a file without actually reading or writing.

```
seek( fd, offset, ptr );
```

forces the current position in the file whose descriptor is fd to move to position offset, which is taken with respect to the location in the file specified by ptr. ptr can be 0, 1, or 2 to specify an offset measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a newly-opened file,

```
seek( fd, 0, 2 );
```

and to get back to the beginning ("rewind"),

```
seek( fd, 0, 0 );
```

With seek, it is possible to treat files more or less like large arrays, at the price of slower access. Here is a routine to read an arbitrary record from an arbitrary place in a file.

```
get( fd, pos, buf, n )
int fd, pos, n;
char *buf;
{
        seek( fd, pos, 0 );     /* get to pos */
        n = read( fd, buf, n );
        return( n );
}
```

Since integers have only 16 bits, the **offset** specified is limited to 65,536; for this reason, **ptr** values of 3, 4, 5 cause **seek** to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret **ptr** as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file you need two seeks, first one which selects the block, then one which has **ptr** equal to 1 and moves to the desired byte within the block.

6.3. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of −1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell **errno**. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine **perror** will print a message associated with the value of **errno**; more generally, **sys_ermo** is an array of character strings which can be indexed by **errno** and printed by your program.

7. Executing Commands From Programs

It is often easier to use a program written by someone else instead of inventing one's own. This section describes how to call a command from within a running program.

7.1. With the Portable C Library

The portable C library routine **system** takes one argument, a command string exactly as you would have typed it at the terminal (except for the new-line at the end) and executes it. This is probably the easiest way to execute a command from within a running program. For instance, to time-stamp the output of a program,

```
main( )
{
        system( "date" );
        /* rest of processing */
}
```

If the command string has to be built from pieces, the core-to-core formatting capabilities of **printf** may be useful (in the portable library only).

7.2. With Standard UNIX

If you're not using the portable C library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the portable library's **system** routine is based on. For no good reason, the standard library doesn't have an equivalent of **system**.

First, you can execute another program *without returning*, by using the routine **execl**, (described under *exec* in Section II of the manual). To print the date as the last action of a running program, you can say

```
execl( "/bin/date", "date", 0 );
```

The first argument to **execl** is the *file name* of the command; you have to know where it's found. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this, and the whole list is followed by a 0 to terminate it.

The **execl** call overlays your program with **date**, runs it, then exits. More realistically, your program might fall into two or more phases that communicate only through temporary files, like the assembler. Here it is natural to make the second pass simply an **execl** call from the first.

The one exception to the statement that your program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where **date** is located, say

```
execl( "/bin/date", "date", 0 );
execl( "/usr/bin/date", "date", 0 );
printf( "Someone stole 'date'\n", 0 );
```

Another version of *exec*, **execv**, is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv( filename, argp );
```

where **argp** is a list of pointers to the arguments; the last pointer must be followed by a 0 so **execv** can tell where the list ends. As with **execl**, **filename** is the file in which the program is found, and **argp[0]** is the name of the program.

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories— you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like '<', '>', '*', '?', and '[ ]' in the argument list. If you want these, use **execl** to invoke the shell **sh**, which then does all the work. Construct a string that contains the complete command you would have typed at the terminal, then say

```
execl( "/bin/sh", "sh", "-c", commandline, 0 );
```

The shell is always going to be at a fixed place, */bin/sh*. Its argument "-c" says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in *commandline*.

7.3. Regaining Control

So far what we've talked about isn't really all that useful by itself. Now we show how to regain control after running a program with **execl** or **execv**. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by the routine called **fork**:

```
pid = fork( );
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of **pid**. In one of these processes (the "child"), **pid** is zero; in the other (the "parent"), **pid** is non-zero; it is the process name of the child. Thus the basic way to call, and return from, another program is

```
if ( (fork( ) == 0 )
        execl( "/bin/sh", "sh", "-c", command, 0 );
```

And in fact, except for handling errors, this is sufficient. The **fork** makes two copies of your program. In the child, the value returned by **fork** is zero, so it calls **execl** which does the **command** and then dies. In the parent, **fork** returns non-zero so it skips the **execl**.

More often, the parent wants to wait for the child to terminate, so output doesn't get scrambled. This can be done with

```
if ( fork( ) == 0 )
        execl( ... );
wait( &status );
```

This still doesn't handle any abnormal conditions, such as a failure of the **execl** or **fork**, or the possibility that there might be more than one child running simultaneously. (The **wait** returns the **pid** of the terminated child, if you want to check it against the value returned by **fork**.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in **status**). Still, these three lines are the heart of the portable library's **system** routine.

The status return word set by **wait** encodes in its low-order byte the system's idea of the child's termination status; it is 0 for OK and non-zero to indicate various kinds of problems like those mentioned in section 8. The high-order byte is taken from the argument of the call to [c]exit which caused a normal termination of the child process. At the moment, the standard command interpreter (the shell) isn't fussy about termination status, but it is good coding practice for all programs to return meaningful status; someday, they may be called by another program which cares whether they worked right.

When your program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When you call another program, correct etiquette suggests making sure the same conditions hold. Neither of the **exec** calls affects open files in any way. Remember too, that both **fork** and **exec** create processes whose address space is distinct from that of their caller. Some buffer-flushing may be needed before using these calls. Conversely, if a caller buffers an input stream, the callee will lose the read-ahead information. (Essentially the same syndrome was discussed in §6.1.)

8. Signals — Interrupts and all that

This section is concerned with how you can make your program deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: **interrupt**, which is sent when the DEL character is typed; **quit**, generated by the FS character; and **hangup**, caused by hanging up the phone. When one of these events occurs, the signal is sent to all processes which were started from the corresponding typewriter; unless other arrangements have been made, the signal terminates the process. In the **quit** case, a core image file is written, usually for debugging purposes.

The routine which alters the default action is **signal**, described in section II of [1]. It has two arguments: the first names the signal, the second specifies how to treat it. If the second argument is 1, the signal is ignored; if it is 0, the default action is restored. Thus

```
#define SIGINT 2
 ...
signal(SIGINT, 1);
```

ignores interrupts, while

```
signal(SIGINT, 0);
```

restores the default action of process termination. Such coding is seldom needed (though see below) because there is a command which runs another program with these three signals ignored:

```
nohup program &
```

runs **program** (with arguments if you like) in such a way that you can hang up on it without fear. Usually you would follow the command with an "&"; otherwise your terminal will be

firmly tied up. If the command is going to run for a long time, you might also use **nice**:

```
nice nohup program &
```

Nice lowers the priority of **program** so it won't hog the machine. Incidentally, starting a command with the "&" automatically causes interrupts and quits to be ignored, so you can compute in the background, and edit and debug in the foreground without danger. However, hangups will still terminate "&" programs.

Finally, the second argument to **signal** may be the name of a function (which, incidentally, has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
main( )
{
        extern int onintr( );
        if ( (signal(SIGINT, 1) & 1) == 0 )
                signal( SIGINT, onintr );
        /* Process ... */
}

onintr( )
{
        unlink( tempfile );
        exit( 100 );
}
```

Why the test and the double call to **signal**? It's quite simple: suppose this "interactive" program were run non-interactively, say under "&" with input from a file. If it began by announcing that all interrupts were to be sent to the **onintr** routine, that would in fact occur, even if the user meant to interrupt only some foreground process he happened to be running. The code as written depends on the fact that **signal** returns the previous value of its argument; the **if** clause asks whether interrupt was previously being ignored (value is odd, e.g. 1) and only if not does it request the call to **onintr**. In other words, if interrupts were being ignored when the program was called, they should still be ignored.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of the editor; interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
main( ) {
        extern int onintrup( );
        setexit( );
        if ( (signal(SIGINT, 1) & 1) == 0 )
                signal( SIGINT, onintrup );
        for ( ; ; ) {
        /* main processing loop */
        }
```

```
onintrup( )
{
        printf( "\nInterrupt\n" );
        reset( );
}
```

When an interrupt occurs, a call is forced to the **onintrup** routine, which can print a message (and perhaps set flags, etc.). **reset** is a non-local goto to the location after the last call to **setexit**, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that **signal** gets called again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs which want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling **exit** or **reset**, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested at some convenient point in the main loop.

There is one difficulty associated with this approach. Suppose the program is reading the typewriter when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the typewriter until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the typewriter read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for 'errors' which are caused by interrupted system calls. (The ones to watch out for are reads from a typewriter, **wait**, and **sleep**.) A program whose **onintrup** program just sets **intflag**, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if ( getchar( ) == '\0' )
        if ( intflag )
                /* interrupt processing */
        else
                /* end-of-file or error processing */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose your program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Your code should look something like this:

```
signal( SIGINT, 1 );    /* ignore interrupts */
if ( fork( ) == 0 )
        execl( ... );
wait( ... );
signal( SIGINT, onintrup );     /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your typewriter. But so also will the calling program pop out of its wait for the subprogram, and also read your typewriter. Having two processes reading

your typewriter is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done.

References

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition. Bell Laboratories, 1975.

[2] B. W. Kernighan, *Programming in C — A Tutorial.* In *The Programming Language C*, Bell Laboratories Computer Science Technical Report 31 (1975).

[3] B. W. Kernighan, *UNIX For Beginners.* Bell Laboratories internal memorandum.

[4] M. E. Lesk, *The Portable C Library.* In *The Programming Language C*, Bell Laboratories Computer Science Technical Report 31 (1975).

C Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

May 1, 1977

1. Introduction

C is a computer language which offers a rich selection of operators and data types and the ability to impose useful structure on both control flow and data. All the basic operations and data objects are close to those actually implemented by most real computers, so that a very efficient implementation is possible, but the design is not tied to any particular machine and with a little care it is possible to write easily portable programs.

This manual describes the current version of the C language as it exists on the PDP-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

Blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. Some space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore '_' counts as alphabetic. Upper and lower case letters are considered different. On the PDP-11, no more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

Warning: The data type name **short** is *not* recognized by the version of the C compiler that is distributed as part of PWB/UNIX Edition 1.0.

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

The entry keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the word fortran.

2.4 Constants

There are several kinds of constants, as follows:

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. The digits 8 and 9 have octal value 10 and 11 respectively. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer (32767 on the PDP-11) is taken to be long; an octal or hex constant which exceeds the largest unsigned machine integer (0177777 or 0xFFFF on the PDP-11) is likewise taken to be long.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by l (letter ell) or L is a long constant, which, on the PDP-11, has 32 significant bits. As discussed below, on other machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a sequence of characters enclosed in single quotes ''. Within a character constant a single quote must be preceded by a backslash '\'. Certain non-graphic characters, and '\' itself, may be escaped according to the following table:

BS	\b
NL (LF)	\n
CR	\r
HT	\t
FF	\f
ddd	*ddd*
\	\\

The escape '*ddd*' consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is '\0' (not followed by a digit) which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash vanishes.

The value of a single-character constant is the numerical value of the character in the machine's character set (ASCII for the PDP-11). On the PDP-11 at most two characters are permitted in a character constant and the second character of a pair is stored in the high-order byte of the integer value. Character constants with more than one character are inherently machine-dependent and should be avoided.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

A string is a sequence of characters surrounded by double quotes '"'. A string has type 'array of characters' and storage class 'static' (see below) and is initialized with the given characters. The compiler places a null byte '\0' at the end of each string so that programs which scan the string can find its end. In a string, the character '"' must be preceded by a '\'; in addition, the same escapes as described for character constants may be used. Finally, a '\' and an immediately following new-line are ignored.

All strings, even when written identically, are distinct.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in sans-serif type. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript 'opt,' so that

$$\{\ expression_{opt}\ \}$$

would indicate an optional expression in braces. The complete syntax is given in §16, in the notation of YACC.

4. What's in a Name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block, and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (char) are large enough to store any member of the implementation's character set, and if a genuine character is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent. On the PDP-11, characters are stored as signed 8-bit integers, and the character set is ASCII.

Up to three sizes of integer, declared short int, int, and long int are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both equivalent to plain integers. 'Plain' integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs. On the PDP-11, short and plain integers are both represented in 16-bit 2's complement notation. Long integers are 32-bit 2's complement.

Unsigned integers, declared unsigned, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (16 on the PDP-11; long and short unsigned quantities

are not supported.)

Single precision floating point (float) quantities, on the PDP-11, have magnitude in the range approximately $10^{\pm 38}$ or 0; their precision is 24 bits or about seven decimal digits.

Double-precision floating-point (double) quantities on the PDP-11 have the same range as floats and a precision of 56 bits or about 17 decimal digits. Some implementations may make float and double synonymous.

Because objects of these types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types char and int of all sizes will collectively be called *integral* types. Float and double will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;

functions which return objects of a given type;

pointers to objects of a given type;

structures containing a sequence of objects of various types;

unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name 'lvalue' comes from the assignment expression '*E*1 = *E*2' in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a short integer always involves sign extension; short integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. On the PDP-11, character variables range in value from −128 to 127; a character constant specified using an octal escape also suffers sign extension and may appear negative, for example ''\214''.

When a longer integer is converted to a shorter or to a char, it is truncated on the left.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a float appears in an expression it is lengthened to double by zero-padding its fraction. When a double must be converted to float, for example by an assignment, the double is rounded before truncation to float length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent. On the PDP-11, truncation is towards 0. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value (on the PDP-11) is the least unsigned integer congruent to the signed integer (modulo 2^{16}). Because of the 2's complement notation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to long, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the 'usual arithmetic conversions.'

First, any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.

Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.

Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.

Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.

Otherwise, both operands must be **int**, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the collected grammar.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. Expressions involving a commutative and associative operator may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

7.1 Primary expressions

Primary expressions involving ., −>, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(*expression*)
primary-expression [*expression*]
primary-expression ($expression\text{-}list_{opt}$)
primary-lvalue . *identifier*
primary-expression −> *identifier*

expression-list:
expression
expression-list , *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is 'array of ...', then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is 'pointer to ...'. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared 'function returning ...', when used except in the function-name position of a call, is converted to 'pointer to function returning ...'.

A decimal, octal, character, or floating constant is a primary expression. Its type may be int, long, or double depending on its form.

A string is a primary expression. Its type is originally 'array of char'; but following the same rule given above for identifiers, this is modified to 'pointer to char' and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type 'pointer to ...', the subscript expression is int, and the type of the result is '...'. The expression 'E1[E2]' is identical (by definition) to '*((E1)+(E2))'. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type 'function returning ...', and the result of the function call is of type '...'. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type float are converted to double before the call; any of type char or short are converted to int.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a '−' and a '>') followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression 'E1−>MOS' is the same as '(*E1).MOS'. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

unary-expression:
- * *expression*
- & *lvalue*
- − *expression*
- ! *expression*
- ~ *expression*
- ++ *lvalue*
- −− *lvalue*
- *lvalue* ++
- *lvalue* −−
- (*type-name*) *expression*
- sizeof *expression*
- sizeof (*type-name*)

The unary * operator means *indirection:* the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is 'pointer to ...', the type of the result is '...'.

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is '...', the type of the result is 'pointer to ...'.

The result of the unary − operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is 16 on the PDP-11.

The result of the logical negation operator ! is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is int. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix '++' is incremented. The value is the new value of the operand, but is not an lvalue. The expression '++a' is equivalent to '(a += 1)'. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix '−−' is decremented analogously to the ++ operator.

When postfix '++' is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix '−−' is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the

prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. The construction of type names is described in §8.7.

The sizeof operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of sizeof. However in all existing implementations a byte is the space required to hold a char.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The sizeof operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction 'sizeof(type)' is taken to be a unit, so the expression 'sizeof(type)−2' is the same as '(sizeof(type))−2'.

7.3 Multiplicative operators

The multiplicative operators *, /, and % group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:
expression * *expression*
expression / *expression*
expression % *expression*

The binary * operator indicates multiplication. The * operator is associative and expressions with several multiplications at the same level may be rearranged.

The binary / operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. In all cases it is true that $(a/b)*b + a\%b = a$. On the PDP-11, the remainder has the same sign as the dividend.

The binary % operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. On the PDP-11, the remainder has the same sign as the dividend. The operands must not be floating.

7.4 Additive operators

The additive operators + and − group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression − *expression*

The result of the '+' operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if P is a pointer to an object in an array, the expression 'P+1' is a pointer to the next object in the array.

No further type combinations are allowed.

The + operator is associative and expressions with several additions at the same level may be rearranged.

The result of the '−' operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a

pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an int representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to int; the type of the result is that of the left operand. The result is undefined if the right operand is negative or larger than the number of bits in the object.

shift-expression:
expression << *expression*
expression >> *expression*

The value of 'E1<<E2' is E1 (interpreted as a bit pattern) left-shifted E2 bits; vacated bits are 0-filled. The value of 'E1>>E2' is E1 right-shifted E2 bit positions. The shift is guaranteed to be logical (0-fill) if E1 is unsigned; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; 'a<b<c' does not mean what it seems to.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int. The usual arithmetic conversions are performed. Two pointers may be compared, and the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus 'a<b == c<d' is 1 whenever a<b and c<d have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise *and* operator

and-expression:
expression & *expression*

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bit-wise 'and' function of the operands. The operator applies only to integral operands.

7.9 Bitwise *exclusive or* operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is is the bit-wise 'exclusive or' function of the operands. The operator applies only to integral operands.

7.10 Bitwise *inclusive or* operator

inclusive-or-expression:
expression | *expression*

The | operator is associative and expressions with | may be rearranged. The usual arithmetic conversions are performed; the result is the bit-wise 'inclusive or' function of its operands. The operator applies only to integral operands.

7.11 Logical *and* operator

logical-and-expression:
expression && *expression*

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always int.

7.12 Logical *or* operator

logical-or-expression:
expression || *expression*

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always int.

7.13 Conditional operator

conditional-expression:
expression ? *expression* : *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = *expression*
lvalue + = *expression*
lvalue − = *expression*
lvalue * = *expression*
lvalue / = *expression*
lvalue % = *expression*
lvalue > > = *expression*
lvalue < < = *expression*
lvalue & = *expression*
lvalue ^ = *expression*
lvalue | = *expression*

Notice that the representation of the compound assignment operators has changed; formerly the '=' came first and the other operator came second (without any space). The compiler continues to accept the previous notation.

In the simple assignment with '=', the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment.

The behavior of an expression of the form 'E1 op = E2' may be inferred by taking it as equivalent to 'E1 = E1 op (E2)'; however, E1 is evaluated only once. In + = and − =, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compiler currently allows a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

7.15 Comma operator

comma-expression:
expression , *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example, 'f(a, (t = 3, t+2), c)' has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used within function definitions to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The typedef specifier does not reserve storage and is called a 'storage class specifier' only for syntactic convenience; it is discussed in §8.8.

The meanings of the various storage classes were discussed in §4.

The auto, static, and register declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the extern case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A register declaration is best thought of as an auto declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few (three, for the PDP-11) such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are int, char, or pointer. One restriction applies to register variables: the address-of operator & cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future developments may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be auto inside a function, extern outside. Exception: functions are always extern.

8.2 Type specifiers

The type-specifiers are

type-specifier:
char
short
int
long
unsigned
float
double
struct-or-union-specifier
typedef-name

The words long, short, and unsigned may be thought of as adjectives; the following combinations are acceptable (in any order).

```
short int
long int
unsigned int
long float
```

The meaning of the last is the same as double. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be int.

Specifiers for structures and unions are discussed in §8.5; declarations with typedef names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , *declarator-list*

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(*declarator*)
* *declarator*
declarator ( )
declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

If a declarator has the form

```
* D
```

for D a declarator, then the contained identifier has the type 'pointer to ...', where '...' is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

```
D ( )
```

then the contained identifier has the type 'function returning ...', where '...' is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

```
D[constant-expression]
```

or

D[ ]

Such declarators make the contained identifier have type 'array.' If the unadorned declarator D would specify a non-array of type '...', then the declarator 'D[i]' yields a 1-dimensional array with rank i of objects of type '...'. If the unadorned declarator D would specify an n-dimensional array with rank $i_1 \times i_2 \times \cdots \times i_n$, then the declarator D[$i_{n+1}$] yields an $(n+1)$-dimensional array with rank $i_1 \times i_2 \times \cdots \times i_n \times i_{n+1}$.

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. (Constant expressions are defined precisely in §15.) The constant expression of an array declarator may be missing only for the first dimension. This notation is useful when the array is external and the actual declaration, which allocates storage, is given elsewhere. The constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f( ), *fip( ), (*pfi)( );
```

declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function which returns an integer. It is especially useful to compare the last two. The binding of '*fip()' is '*(fip())', so that the declaration suggests, and the same construction in an expression requires, the calling of a function *fip*, and then using indirection through the (pointer) result to yield an integer. In the declarator '(*pfi)()', the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called.

As another example,

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, *x3d* is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions 'x3d', 'x3d[i]', 'x3d[i][j]', 'x3d[i][j][k]' may reasonably appear in an expression. The first three have type 'array', the last has type int.

8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

structure-or-union-specifier:
struct-or-union { *struct-decl-list* }
struct-or-union identifier { *struct-decl-list* }
struct-or-union identifier

struct-or-union:
    struct
    union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
    struct-declaration
    struct-declaration struct-decl-list

struct-declaration:
    type-specifier struct-declarator-list ;

struct-declarator-list:
    struct-declarator
    struct-declarator , *struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field;* its length is set off from the field name by a colon.

struct-declarator:
    declarator
    declarator : *constant-expression*
    : *constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. On the PDP-11, fields are assigned right-to-left.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The 'next field' presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even int fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

    struct *identifier* { *struct-decl-list* }
    union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

    struct *identifier*
    union *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure or union which contains an instance of itself, as distinct from a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the *count* field of the structure to which *sp* points;

```
s.left
```

refers to the left subtree pointer of the structure *s*. Finally,

```
s.right->tword[0]
```

refers to the first character of the *tword* member of the right subtree of *s*.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by '=', and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }

The '=' is a new addition to the syntax, intended to alleviate potential ambiguities. The current compiler allows it to be omitted when the rest of the initializer is a very simple expression (just a name, string, or constant) or when the rest of the initializer is enclosed in braces.

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving previously declared variables.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate,

written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates. Currently, the PDP-11 compiler also forbids initializing fields in structures.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeding comma-separated list of initializers initialize the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a char array to be initialized by a string. In this case successive members of the string initialize the members of the array.

For example,

```
int x[ ] = { 1, 3, 5 };
```

declares and initializes *x* as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float  y[4][3] = {
       { 1, 3, 5 },
       { 2, 4, 6 },
       { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array *y*[0], namely *y*[0][0], *y*[0][1], and *y*[0][2]. Likewise the next two lines initialize *y*[1] and *y*[2]. The initializer ends early and therefore *y*[3] is initialized with 0. Precisely the same effect could have been achieved by

```
float  y[4][3] = {
       1, 3, 5, 2, 4, 6, 3, 5, 7,
};
```

The initializer for *y* begins with a left brace, but that for *y*[0] does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for *y*[1] and *y*[2]. Also,

```
float  y[4][3] = {
       { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of *y* (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char   msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly, and as an argument of sizeof) it is desired to supply the name of a data type. This is accomplished using a 'type name,' which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
empty
(*abstract-declarator*)
* *abstract-declarator*
abstract-declarator ( )
abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(*abstract-declarator*)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
```

name respectively the types 'integer,' 'pointer to integer,' 'array of 3 pointers to integers,' and 'pointer to an array of 3 integers.' As another example,

```
int i;
...
sin( (double) i);
```

calls the *sin* routine (which accepts a double argument) with an argument appropriately converted.

8.8 Typedef

Declarations whose 'storage class' is typedef do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types. Within the scope of a declaration involving typedef, each of the identifiers appearing as part of any declarators therein become syntactically equivalent to type keywords naming the type associated with the identifiers in the way described in §8.4.

typedef-name:
identifier

For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im;} complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of *distance* is 'int', that of *metricp* is 'pointer to int,' and that of *z* is the specified structure. *Zp* is a pointer to such a structure.

Typedef does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above *distance* is considered to have exactly the same type as any other int variable.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called 'block') is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, at which time it resumes its force.

Any initializations of auto or register variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of static variables are performed only once when the program begins execution. Inside a block, external declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* else *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the 'else' ambiguity is resolved by connecting an else with the last encountered elseless if.

9.4 While statement

The while statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The do statement has the form

do *statement* while (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The for statement has the form

for (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*

This statement is equivalent to

```
expression-1;
while (expression-2) {
        statement
        expression-3;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing expression-2 makes the implied while clause equivalent to 'while(1)'; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The switch statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be int. The statement is typically compound. Any statement within the statement may be labelled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be int. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

default :

When the switch statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a default prefix, control passes to the prefixed statement. If no case matches and if there is no default then none of the statements in the switch is executed.

Case and default prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see break, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

break ;

causes termination of the smallest enclosing while, do, for, or switch statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing while, do, or for statement; that is to the end of the loop. More precisely, in each of the statements

```
while ( ... ) {          do {                    for ( ... ) {
  ...                      ...                     ...
contin: ;                contin: ;               contin: ;
}                        } while ( ... ) ;       }
```

a continue is equivalent to 'goto contin'. (Following the 'contin:' is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (§9.12) located in the current function. Previous versions of C had an incompletely implemented notion of label variable, which has been withdrawn.

9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a goto. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the '}' of a compound statement or to supply a null body to a looping statement such as while.

10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class extern (by default) or perhaps static, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be int. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are extern or static; See §11.2 for the distinction between them. A function declarator is similar to a declarator for a 'function returning ...' except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

The function-body has the form

function-body:
declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be int. The only storage class which may be specified is register; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
        int m;
        m = (a>b)? a:b;
        return(m>c? m:c);
}
```

Here 'int' is the type-specifier; 'max(a, b, c)' is the function-declarator; 'int a, b, c;' is the declaration-list for the formal parameters; '{ ... }' is the block giving the code for the statement. The parentheses in the return are not required.

C converts all float actual parameters to double, so formal parameters declared float have their declaration adjusted to read double. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared 'array of ...' are adjusted to read 'pointer to ...'. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free return statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

10.2 External data definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be extern (which is the default) or static, but not auto or register.

11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing 'undefined identifier' diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Typedef names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{   auto int distance;
    ...
```

The int must be present in the second declaration, or it would be taken to be a declaration with no declarators and type distance.*

11.2 Scope of externals

If a function declares an identifier to be extern, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and extent specified in the definition are compatible with those specified by each function which references the data.

In PDP-11 C, compatible external definitions of the same identifier may be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. The appearance of the extern keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an

*It is agreed that the ice is thin here.

external data definition without the extern specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the extern in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared static at the top level in external definitions are not visible in other files.

12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with '#' communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

define *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

define *identifier*(*identifier* , ... , *identifier*) *token-string*

where there is no space between the first identifier and the '(', is a macro definition with arguments. Subsequent instances of the first identifier followed by a '(', a sequence of tokens delimited by commas, and a ')' are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing '\' at the end of the line to be continued.

This facility is most valuable for definition of 'manifest constants', as in

```
# define TABSIZE 100
...
int table[TABSIZE];
```

A control line of the form

undef *identifier*

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

include "*filename*"

causes the replacement of that line by the entire contents of the file *filename*.

The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

include <*filename*>

searches only the standard places, and not the directory of the source file.

Includes may be nested.

12.3 Conditional compilation

A compiler control line of the form

if *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a #define control line. A control line of the form

ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

else

and then by a control line

endif

If the checked condition is true then any lines between #else and #endif are ignored. If the checked condition is false then any lines between the test and an #else or, lacking an #else, the #endif, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

line *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the next line number is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes the storage class is supplied by the context: in external definitions, and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be int; if a type but no storage class is indicated, the identifier is assumed to be auto. An exception to the latter rule is made for functions, since auto functions are meaningless (C being incapable of compiling code into the stack). If the type of an identifier is 'function returning ...', it is implicitly declared to be extern.

In an expression, an identifier followed by (and not currently declared is contextually declared to be 'function returning int'.

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the . operator); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with . or −>) the name on the

right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before '.', and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a '->' is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of *g* might read

```
g(funcp)
int (*funcp)();
{
    ...
    (*funcp)();
    ...
}
```

Notice that *f* was declared explicitly in the calling routine since its first appearance was not followed by (.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that 'E1[E2]' is identical to '*((E1) + (E2))'. Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If E is an n-dimensional array of rank $i \times j \times \cdots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here x is a 3×5 array of integers. When x appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression 'x[i]', which is equivalent to '*(x+i)', x is first converted to a pointer as described; then i is converted to the type of x, which involves multiplying i by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed

by an array but plays no other part in subscript calculations.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after case, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and sizeof expressions, possibly connected by the binary operators

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  <  >  <=  >=
```

or by the unary operators

```
-  ~
```

or by the ternary operator

```
? :
```

Parentheses can be used for grouping, but not for function calls.

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary & operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

16. Grammar revisited.

This section repeats the grammar of C in notation somewhat different than given before. The description below is adapted directly from a YACC grammar actually used by several compilers; thus it may (aside from possible editing errors) be regarded as authentic. The notation is pure YACC with the exception that the '|' separating alternatives for a production is omitted, since alternatives are always on separate lines; the ';' separating productions is omitted since a blank line is left between productions.

The lines with '%term' name the terminal symbols, which are either commented upon or should be self-evident. The lines with '%left,' '%right,' and '%binary' indicate whether the listed terminals are left-associative, right-associative, or non-associative, and describe a precedence structure. The precedence (binding strength) increases as one reads down the page. When the construction '%prec *x*' appears the precedence of the rule is that of the terminal *x*; otherwise the precedence of the rule is that of its leftmost terminal.

```
%term NAME
%term STRING
%term ICON
%term FCON
%term PLUS
%term MINUS
%term MUL
%term AND
%term QUEST
%term COLON
%term ANDAND
%term OROR
%term ASOP      /* old-style =+ etc. */
%term RELOP     /* <= >= < > */
%term EQUOP     /* == != */
%term DIVOP     /* / % */
%term OR        /* | */
%term EXOR      /* ^ */
%term SHIFTOP   /* << >> */
%term INCOP     /* ++ -- */
%term UNOP      /* ! ~ */
%term STROP     /* . -> */


%term TYPE      /* int, char, long, float, double, unsigned, short */
%term CLASS     /* extern, register, auto, static, typedef */
%term STRUCT    /* struct or union */
%term RETURN
%term GOTO
%term IF
%term ELSE
%term SWITCH
%term BREAK
%term CONTINUE
%term WHILE
%term DO
%term FOR
%term DEFAULT
%term CASE
%term SIZEOF
```

```
%term LP        /* ( */
%term RP        /* ) */
%term LC        /* { */
%term RC        /* } */
%term LB        /* [ */
%term RB        /* ] */
%term CM        /* , */
%term SM        /* ; */
%term ASSIGN    /* = */


%left   CM
%right  ASOP    ASSIGN
%right  QUEST   COLON
%left   OROR
%left   ANDAND
%left   OROP
%left   AND
%binary EQUOP
%binary RELOP
%left   SHIFTOP
%left   PLUS    MINUS
%left   MUL     DIVOP
%right  UNOP
%right  INCOP   SIZEOF
%left   LB      LP      STROP


program:        ext_def_list

ext_def_list:   ext_def_list external_def
                /* empty */

external_def:   optattrib SM
                optattrib init_dcl_list SM
                optattrib fdeclarator function_body

function_body:  dcl_list compoundstmt

dcl_list:       dcl_list declaration
                /* empty */

declaration:    specifiers declarator_list SM
                specifiers SM

optattrib:      specifiers
                /* empty */

specifiers:     CLASS type
                type CLASS
                CLASS
                type
```

```
type:           TYPE
                TYPE TYPE
                struct_dcl

struct_dcl:     STRUCT NAME LC type_dcl_list RC
                STRUCT LC type_dcl_list RC
                STRUCT NAME

type_dcl_list:  type_declaration
                type_dcl_list type_declaration

type_declaration: type declarator_list SM
                struct_dcl SM
                type SM

declarator_list: declarator
                declarator_list CM declarator

declarator:     fdeclarator
                nfdeclarator
                nfdeclarator COLON con_e %prec CM
                COLON con_e %prec CM

nfdeclarator:   MUL nfdeclarator
                nfdeclarator LP RP
                nfdeclarator LB RB
                nfdeclarator LB con_e RB
                NAME
                LP nfdeclarator RP

fdeclarator:    MUL fdeclarator
                fdeclarator LP RP
                fdeclarator LB RB
                fdeclarator LB con_e RB
                LP fdeclarator RP
                NAME LP name_list RP
                NAME LP RP

name_list:      NAME
                name_list CM NAME

init_dcl_list:  init_declarator %prec CM
                init_dcl_list CM init_declarator

init_declarator: nfdeclarator
                nfdeclarator ASSIGN initializer
                nfdeclarator initializer
                fdeclarator

init_list:      initializer %prec CM
                init_list CM initializer

initializer:    e %prec CM
                LC init_list RC
```

```
                LC init_list CM RC

compoundstmt:   LC dcl_list stmt_list RC

stmt_list:      stmt_list statement
                /* empty */

statement:      e SM
                compoundstmt
                IF LP e RP statement
                IF LP e RP statement ELSE statement
                WHILE LP e RP statement
                DO statement WHILE LP e RP SM
                FOR LP opt_e SM opt_e SM opt_e RP statement
                SWITCH LP e RP statement
                BREAK SM
                CONTINUE SM
                RETURN SM
                RETURN e SM
                GOTO NAME SM
                SM
                label statement

label:          NAME COLON
                CASE con_e COLON
                DEFAULT COLON

con_e:          e %prec CM

opt_e:          e
                /* empty */

elist:          e %prec CM
                elist CM e

e:              e MUL e
                e CM e
                e DIVOP e
                e PLUS e
                e MINUS e
                e SHIFTOP e
                e RELOP e
                e EQUOP e
                e AND e
                e OROP e
                e ANDAND e
                e OROR e
                e MUL ASSIGN e
                e DIVOP ASSIGN e
                e PLUS ASSIGN e
                e MINUS ASSIGN e
                e SHIFTOP ASSIGN e
                e AND ASSIGN e
                e OROP ASSIGN e
```

```
                e QUEST e COLON e
                e ASOP e
                e ASSIGN e
                term

term:           term INCOP
                MUL term
                AND term
                MINUS term
                UNOP term
                INCOP term
                SIZEOF term
                LP type_name RP term %prec STROP
                SIZEOF LP type_name RP %prec SIZEOF
                term LB e RB
                term LP RP
                term LP elist RP
                term STROP NAME
                NAME
                ICON
                FCON
                STRING
                LP e RP

type_name:      type abst_decl

abst_decl:      /* empty */
                LP RP
                LP abst_decl RP LP RP
                MUL abst_decl
                abst_decl LB RB
                abst_decl LB con_e RB
                LP abst_decl RP
```

Programming in C — A Tutorial

Brian W. Kernighan

Bell Laboratories, Murray Hill, N. J.

1. Introduction

C is a computer language available on the GCOS and UNIX operating systems at Murray Hill and (in preliminary form) on OS/360 at Holmdel. C lets you write your programs clearly and simply — it has decent control flow facilities so your code can be read straight down the page, without labels or GOTO's; it lets you write code that is compact without being too cryptic; it encourages modularity and good program organization; and it provides good data-structuring facilities.

This memorandum is a tutorial to make learning C as painless as possible. The first part concentrates on the central features of C; the second part discusses those parts of the language which are useful (usually for getting more efficient and smaller code) but which are not necessary for the new user. This is *not* a reference manual. Details and special cases will be skipped ruthlessly, and no attempt will be made to cover every language feature. The order of presentation is hopefully pedagogical instead of logical. Users who would like the full story should consult the *C Reference Manual* by D. M. Ritchie [1], which should be read for details anyway. Runtime support is described in [2] and [3]; you will have to read one of these to learn how to compile and run a C program.

We will assume that you are familiar with the mysteries of creating files, text editing, and the like in the operating system you run on, and that you have programmed in some language before.

2. A Simple C Program

```
main( ) {
        printf("hello, world");
}
```

A C program consists of one or more *functions*, which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. **main** is such a function, and in fact all C programs must have a **main**. Execution of the program begins at the first statement of **main**. **main** will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here **main** is a function of no arguments, indicated by ( ). The {} enclose the statements of the function. Individual statements end with a semicolon but are otherwise free-format.

printf is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A function is invoked by naming it, followed by a list of arguments in parentheses. There is no CALL statement as in Fortran or PL/I.

3. A Working C Program; Variables; Types and Type Declarations

Here's a bigger program that adds three integers and prints their sum.

```
main( ) {
        int a, b, c, sum;
        a = 1;  b = 2;  c = 3;
        sum = a + b + c;
        printf("sum is %d", sum);
}
```

Arithmetic and the assignment statements are much the same as in Fortran (except for the semicolons) or PL/I. The format of C programs is quite free. We can put several statements on a line if we want, or we can split a statement among several lines if it seems desirable. The split may be between any of the operators or variables, but *not* in the middle of a name or operator. As a matter of style, spaces, tabs, and newlines should be used freely to enhance readability.

C has four fundamental *types* of variables:

```
int     integer (PDP-11: 16 bits; H6070: 36 bits; IBM360: 32 bits)
char    one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
float   single-precision floating point
double  double-precision floating point
```

There are also *arrays* and *structures* of these basic types, *pointers* to them and *functions* that return them, all of which we will meet shortly.

All variables in a C program must be declared, although this can sometimes be done implicitly by context. Declarations must precede executable statements. The declaration

```
int a, b, c, sum;
```

declares a, b, c, and sum to be integers.

Variable names have one to eight characters, chosen from A-Z, a-z, 0-9, and _, and start with a non-digit. Stylistically, it's much better to use only a single case and give functions and external variables names that are unique in the first six characters. (Function and external variable names are used by various assemblers, some of which are limited in the size and case of identifiers they can handle.) Furthermore, keywords and library functions may only be recognized in one case.

4. Constants

We have already seen decimal integer constants in the previous example — 1, 2, and 3. Since C is often used for system programming and bit-manipulation, octal numbers are an important part of the language. In C, any number that begins with 0 (zero!) is an octal integer (and hence can't have any 8's or 9's in it). Thus 0777 is an octal constant, with decimal value 511.

A "character" is one byte (an inherently machine-dependent concept). Most often this is expressed as a *character constant*, which is one character enclosed in single quotes. However, it may be any quantity that fits in a byte, as in flags below:

```
char quest, newline, flags;
quest = '?';
newline = '\n';
flags = 077;
```

The sequence '\n' is C notation for "newline character", which, when printed, skips the terminal to the beginning of the next line. Notice that '\n' represents only a single character. There are several other "escapes" like '\n' for representing hard-to-get or invisible characters, such as '\t' for tab, '\b' for backspace, '\0' for end of file, and '\\' for the backslash itself.

float and **double** constants are discussed in section 26.

5. Simple I/O — getchar, putchar, printf

```
main( ) {
        char c;
        c = getchar( );
        putchar(c);
}
```

getchar and **putchar** are the basic I/O library functions in C. **getchar** fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by '\0' (ascii NUL, which has value zero). We will see how to use this very shortly.

putchar puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

printf is a more complicated function for producing formatted output. We will talk about only the simplest use of it. Basically, **printf** uses its first argument as formatting information, and any successive arguments as variables to be output. Thus

```
printf ("hello, world\n");
```

is the simplest use — the string "hello, world\n" is printed out. No formatting information, no variables, so the string is dumped out verbatim. The newline is necessary to put this out on a line by itself. (The construction

```
"hello, world\n"
```

is really an array of **chars**. More about this shortly.)

More complicated, if **sum** is 6,

```
printf ("sum is %d\n", sum);
```

prints

```
sum is 6
```

Within the first argument of **printf**, the characters "%d" signify that the next argument in the argument list is to be printed as a base 10 number.

Other useful formatting commands are "%c" to print out a single character, "%s" to print out an entire string, and "%o" to print a number as octal instead of decimal (no leading zero). For example,

```
n = 511;
printf ("What is the value of %d in octal?", n);
```

```
printf ("  %s! %d decimal is %o octal\n", "Right", n, n);
```

prints

```
What is the value of 511 in octal?  Right! 511 decimal is 777 octal
```

Notice that there is no newline at the end of the first output line. Successive calls to printf (and/or putchar, for that matter) simply put out characters. No newlines are printed unless you ask for them. Similarly, on input, characters are read one at a time as you ask for them. Each line is generally terminated by a newline (\n), but there is otherwise no concept of record.

6. If; relational operators; compound statements

The basic conditional-testing statement in C is the if statement:

```
c = getchar( );
if( c == '?' )
        printf("why did you type a question mark?\n");
```

The simplest form of if is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. There's an optional **else** clause, to be described soon.

The character sequence '==' is one of the relational operators in C; here is the complete set:

```
==    equal to (.EQ. to Fortraners)
!=    not equal to
>     greater than
<     less than
>=    greater than or equal to
<=    less than or equal to
```

The value of "expression relation expression" is 1 if the relation is true, and 0 if false. Don't forget that the equality test is '=='; a single '=' causes an assignment, not a test, and invariably leads to disaster.

Tests can be combined with the operators '&&' (AND), '||' (OR), and '!' (NOT). For example, we can test whether a character is blank or tab or newline with

```
if( c==' ' || c=='\t' || c=='\n' ) ...
```

C guarantees that '&&' and '||' are evaluated left to right — we shall soon see cases where this matters.

One of the nice things about C is that the statement part of an if can be made arbitrarily complicated by enclosing a set of statements in {}. As a simple example, suppose we want to ensure that a is bigger than b, as part of a sort routine. The interchange of a and b takes three statements in C, grouped together by {}:

```
if (a < b) {
        t = a;
        a = b;
        b = t;
}
```

As a general rule in C, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in {}. There is no semicolon after the } of a compound statement, but there *is* a semicolon after the last non-compound statement inside the {}.

The ability to replace single statements by complex ones at will is one feature that makes C much more pleasant to use than Fortran. Logic (like the exchange in the previous example) which would require several GOTO's and labels in Fortran can and should be done in C without any, using compound statements.

7. While Statement; Assignment within an Expression; Null Statement

The basic looping mechanism in C is the **while** statement. Here's a program that copies its input to its output a character at a time. Remember that '\0' marks the end of file.

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
                putchar(c);
}
```

The **while** statement is a loop, whose general form is

```
while (expression) statement
```

Its meaning is

```
(a) evaluate the expression
(b) if its value is true (i.e., not zero)
                do the statement, and go back to (a)
```

Because the expression is tested before the statement is executed, the statement part can be executed zero times, which is often desirable. As in the if statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. Our example gets the character, assigns it to c, and then tests if it's a '\0'. If it is not a '\0', the statement part of the **while** is executed, printing the character. The **while** then repeats. When the input character is finally a '\0', the **while** terminates, and so does **main**.

Notice that we used an assignment statement

```
c = getchar( )
```

within an expression. This is a handy notational shortcut which often produces clearer code. (In fact it is often the only way to write the code cleanly. As an exercise, re-write the file-copy without using an assignment inside an expression.) It works because an assignment statement has a value, just as any other expression does. Its value is the value of the right hand side. This also implies that we can use multiple assignments like

```
x = y = z = 0;
```

Evaluation goes from right to left.

By the way, the extra parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar( ) != '\0'
```

c would be set to 0 or 1 depending on whether the character fetched was an end of file or not. This is because in the absence of parentheses the assignment operator '=' is evaluated after the relational operator '!='. When in doubt, or even if not, parenthesize.

Since putchar(c) returns c as its function value, we could also copy the input to the output by nesting the calls to getchar and putchar:

```
main( ) {
        while( putchar(getchar( )) != '\0' ) ;
}
```

What statement is being repeated? None, or technically, the *null* statement, because all the work is really done within the test part of the while. This version is slightly different from the previous one, because the final '\0' is copied to the output before we decide to stop.

8. Arithmetic

The arithmetic operators are the usual '+', '−', '*', and '/' (truncating integer division if the operands are both int), and the remainder or mod operator '%':

```
x = a%b;
```

sets x to the remainder after a is divided by b (i.e., a mod b). The results are machine dependent unless a and b are both positive.

In arithmetic, char variables can usually be treated like int variables. Arithmetic on characters is quite legal, and often makes sense:

```
c = c + 'A' - 'a';
```

converts a single lower case ascii character stored in c to upper case, making use of the fact that corresponding ascii letters are a fixed distance apart. The rule governing this arithmetic is that all chars are converted to int before the arithmetic is done. Beware that conversion may involve sign-extension — if the leftmost bit of a character is 1, the resulting integer might be negative. (This doesn't happen with genuine characters on any current machine.)

So to convert a file into lower case:

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
                if( 'A'<=c && c<='Z' )
                        putchar(c+'a'-'A');
                else
                        putchar(c);
}
```

Characters have different sizes on different machines. Further, this code won't work on an IBM machine, because the letters in the ebcdic alphabet are not contiguous.

9. Else Clause; Conditional Expressions

We just used an else after an if. The most general form of if is

```
if (expression) statement1 else statement2
```

the else part is optional, but often useful. The canonical example sets x to the minimum of a and b:

```
if (a < b)
        x = a;
else
        x = b;
```

Observe that there's a semicolon after x=a.

C provides an alternate form of conditional which is often more concise. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. The value of

```
a<b ? a : b;
```

is a if a is less than b; it is b otherwise. In general, the form

```
expr1 ? expr2 : expr3
```

means "evaluate **expr1**. If it is not zero, the value of the whole thing is **expr2**; otherwise the value is **expr3**."

To set x to the minimum of a and b, then:

```
x = (a<b ? a : b);
```

The parentheses aren't necessary because '?:' is evaluated before '=', but safety first.

Going a step further, we could write the loop in the lower-case program as

```
while( (c=getchar( )) != '\0' )
        putchar( ('A'<=c && c<='Z') ? c-'A'+'a' : c );
```

If's and else's can be used to construct logic that branches one of several ways and then rejoins, a common programming structure, in this way:

```
if(...)
        {...}
else if(...)
        {...}
else if(...)
        {...}
else
        {...}
```

The conditions are tested in order, and exactly one block is executed — either the first one whose if is satisfied, or the one for the last **else**. When this block is finished, the next statement executed is the one after the last **else**. If no action is to be taken for the "default" case, omit the last **else**.

For example, to count letters, digits and others in a file, we could write

```
main( ) {
        int let, dig, other, c;
        let = dig = other = 0;
        while( (c=getchar( )) != '\0' )
                if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )  ++let;
                else if( '0'<=c && c<='9' )  ++dig;
                else  ++other;
        printf("%d letters, %d digits, %d others\n", let, dig, other);
}
```

The '++' operator means "increment by 1"; we will get to it in the next section.

10. Increment and Decrement Operators

In addition to the usual '−', C also has two other interesting unary operators, '++' (increment) and '−−' (decrement). Suppose we want to count the lines in a file.

```
main( ) {
        int c,n;
        n = 0;
```

```
        while( (c=getchar( )) != '\0' )
                if( c == '\n' )
                        ++n;
        printf("%d lines\n", n);
}
```

++n is equivalent to n=n+1 but clearer, particularly when n is a complicated expression. '++' and '--' can be applied only to int's and char's (and pointers which we haven't got to yet).

The unusual feature of '++' and '--' is that they can be used either before or after a variable. The value of ++k is the value of k *after* it has been incremented. The value of k++ is k *before* it is incremented. Suppose k is 5. Then

```
x = ++k;
```

increments k to 6 and then sets x to the resulting value, i.e., to 6. But

```
x = k++;
```

first sets x to to 5, and *then* increments k to 6. The incrementing effect of ++k and k++ is the same, but their values are respectively 5 and 6. We shall soon see examples where both of these uses are important.

11. Arrays

In C, as in Fortran or PL/I, it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration

```
int x[10];
```

The square brackets mean *subscripting;* parentheses are used only for function references. Array indexes begin at *zero*, so the elements of x are

```
x[0], x[1], x[2], ..., x[9]
```

If an array has n elements, the largest subscript is n-1.

Multiple-dimension arrays are provided, though not much used above two dimensions. The declaration and use look like

```
int name[10][20];
n = name[i+j][1] + name[k][2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row (opposite to Fortran), so the rightmost subscript varies fastest; name has 10 rows and 20 columns.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {
        int n, c;
        char line[100];
        n = 0;
        while( (c=getchar( )) != '\n' ) {
                if( n < 100 )
                        line[n] = c;
                n++;
        }
        printf("length = %d\n", n);
}
```

As a more complicated problem, suppose we want to print the count for each line in the input, still storing the first 100 characters of each line. Try it as an exercise before looking at the solution:

```
main( ) {
        int n, c; char line[100];
        n = 0;
        while( (c=getchar( )) != '\0' )
                if( c == '\n' ) {
                        printf("%d\n", n);
                        n = 0;
                }
                else {
                        if( n < 100 ) line[n] = c;
                        n++;
                }
}
```

12. Character Arrays; Strings

Text is usually kept as an array of characters, as we did with line[ ] in the example above. By convention in C, the last character in a character array should be a '\0' because most programs that manipulate character arrays expect it. For example, printf uses the '\0' to detect the end of a character array when printing it out with a '%s'.

We can copy a character array s into another t like this:

```
i = 0;
while( (t[i]=s[i]) != '\0' )
        i++;
```

Most of the time we have to put in our own '\0' at the end of a string; if we want to print the line with printf, it's necessary. This code prints the character count before the line:

```
main( ) {
        int n;
        char line[100];
        n = 0;
        while( (line[n++]=getchar( )) != '\n' );
        line[n] = '\0';
        printf("%d:\t%s", n, line);
}
```

Here we increment n in the subscript itself, but only after the previous value has been used. The character is read, placed in line[n], and only then n is incremented.

There is one place and one place only where C puts in the '\0' at the end of a character array for you, and that is in the construction

```
"stuff between double quotes"
```

The compiler puts a '\0' at the end automatically. Text enclosed in double.quotes is called a *string;* its properties are precisely those of an (initialized) array of characters.

13. For Statement

The for statement is a somewhat generalized **while** that lets us put the initialization and increment parts of a loop into a single statement along with the test. The general form of the for is

```
for( initialization; expression; increment )
        statement
```

The meaning is exactly

```
initialization;
while( expression ) {
        statement
        increment;
}
```

Thus, the following code does the same array copy as the example in the previous section:

```
for( i=0; (t[i]=s[i]) != '\0'; i++ );
```

This slightly more ornate example adds up the elements of an array:

```
sum = 0;
for( i=0; i<n; i++)
        sum = sum + array[i];
```

In the for statement, the initialization can be left out if you want, but the semicolon has to be there. The increment is also optional. It is *not* followed by a semicolon. The second clause, the test, works the same way as in the **while**: if the expression is true (not zero) do another loop, otherwise get on with the next statement. As with the **while**, the **for** loop may be done zero times. If the expression is left out, it is taken to be always true, so

```
for( ; ; ) ...
```

and

```
while( 1 ) ...
```

are both infinite loops.

You might ask why we use a for since it's so much like a **while**. (You might also ask why we use a while because...) The for is usually preferable because it keeps the code where it's used and sometimes eliminates the need for compound statements, as in this code that zeros a two-dimensional array:

```
for( i=0; i<n; i++ )
        for( j=0; j<m; j++ )
                array[i][j] = 0;
```

14. Functions; Comments

Suppose we want, as part of a larger program, to count the occurrences of the ascii characters in some input text. Let us also map illegal characters (those with value>127 or <0) into one pile. Since this is presumably an isolated part of the program, good practice dictates making it a separate function. Here is one way:

```
main( ) {
        int hist[129];          /* 128 legal chars + 1 illegal group */
        ...
        count(hist, 128);       /* count the letters into hist */
        printf( ... );          /* comments look like this; use them */
        ...             /* anywhere blanks, tabs or newlines could appear */
}
count(buf, size)
  int size, buf[ ]; {
        int i, c;
        for( i=0; i<=size; i++ )
                buf[i] = 0;                     /* set buf to zero */
        while( (c=getchar( )) != '\0' ) {       /* read till eof */
                if( c > size || c < 0 )
                        c = size;               /* fix illegal input */
                buf[c]++;
        }
        return;
}
```

We have already seen many examples of calling a function, so let us concentrate on how to *define* one. Since count has two arguments, we need to declare them, as shown, giving their types, and in the case of buf, the fact that it is an array. The declarations of arguments go *between* the argument list and the opening '{'. There is no need to specify the size of the array buf, for it is defined outside of count.

The return statement simply says to go back to the calling routine. In fact, we could have omitted it, since a return is implied at the end of a function.

What if we wanted count to return a value, say the number of characters read? The return statement allows for this too:

```
        int i, c, nchar;
        nchar = 0;
        ...
        while( (c=getchar( )) != '\0' ) {
                if( c > size || c < 0 )
                        c = size;
                buf[c]++;
                nchar++;
        }
        return(nchar);
```

Any expression can appear within the parentheses. Here is a function to compute the minimum of two integers:

```
min(a, b)
  int a, b; {
        return( a < b ? a : b );
}
```

To copy a character array, we could write the function

```
strcopy(s1, s2)        /* copies s1 to s2 */
  char s1[ ], s2[ ]; {
       int i;
       for( i = 0; (s2[i] = s1[i]) != '\0'; i++ );
}
```

As is often the case, all the work is done by the assignment statement embedded in the test part of the for. Again, the declarations of the arguments s1 and s2 omit the sizes, because they don't matter to strcopy. (In the section on pointers, we will see a more efficient way to do a string copy.)

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Simple variables (not arrays) are passed in C by "call by value", which means that the called function is given a copy of its arguments, and doesn't know their addresses. This makes it impossible to change the value of one of the actual input arguments.

There are two ways out of this dilemma. One is to make special arrangements to pass to the function the address of a variable instead of its value. The other is to make the variable a global or external variable, which is known to each function by its name. We will discuss both possibilities in the next few sections.

15. Local and External Variables

If we say

```
f( ) {
       int x;
       ...
}
g( ) {
       int x;
       ...
}
```

each x is *local* to its own routine — the x in f is unrelated to the x in g. (Local variables are also called "automatic".) Furthermore each local variable in a routine appears only when the function is called, and *disappears* when the function is exited. Local variables have no memory from one call to the next and must be explicitly initialized upon each entry. (There is a static storage class for making local variables with memory; we won't discuss it.)

As opposed to local variables, *external variables* are defined external to all functions, and are (potentially) available to all functions. External storage always remains in existence. To make variables external we have to *define* them external to all functions, and, wherever we want to use them, make a *declaration*.

```
main( ) {
       extern int nchar, hist[ ];
       ...
       count( );
       ...
}
```

```
count( ) {
        extern int nchar, hist[ ];
        int i, c;
        ...
}

int     hist[129];      /* space for histogram */
int     nchar;          /* character count */
```

Roughly speaking, any function that wishes to access an external variable must contain an **extern** declaration for it. The declaration is the same as others, except for the added keyword **extern**. Furthermore, there must somewhere be a *definition* of the external variables external to all functions.

External variables can be initialized; they are set to zero if not explicitly initialized. In its simplest form, initialization is done by putting the value (which must be a constant) after the definition:

```
int     nchar  0;
char    flag   'f';
  etc.
```

This is discussed further in a later section.

This ends our discussion of what might be called the central core of C. You now have enough to write quite substantial C programs, and it would probably be a good idea if you paused long enough to do so. The rest of this tutorial will describe some more ornate constructions, useful but not essential.

16. Pointers

A *pointer* in C is the address of something. It is a rare case indeed when we care what the specific address itself is, but pointers are a quite common way to get at the contents of something. The unary operator '&' is used to produce the address of an object, if it has one. Thus

```
int a, b;
b = &a;
```

puts the address of a into b. We can't do much with it except print it or pass it to some other routine, because we haven't given b the right kind of declaration. But if we declare that b is indeed a *pointer* to an integer, we're in good shape:

```
int a, *b, c;
b = &a;
c = *b;
```

b contains the address of a and 'c = *b' means to use the value in b as an address, i.e., as a pointer. The effect is that we get back the contents of a, albeit rather indirectly. (It's always the case that '*&x' is the same as x if x has an address.)

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. (You can't change the address of something by assigning to it.) If we say

```
char *y;
char x[100];
```

y is of type pointer to character (although it doesn't yet point anywhere). We can make y point to an element of x by either of

point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0] this is legal and consistent.

Now '*y' gives x[0]. More importantly,

```
*(y+1) gives x[1]
*(y+i) gives x[i]
```

and the sequence

```
y = &x[0];
y++;
```

leaves y pointing at x[1].

Let's use pointers in a function **length** that computes how long a character array is. Remember that by convention all character arrays are terminated with a '\0'. (And if they aren't, this program will blow up inevitably.) The old way:

```
length(s)
   char s[ ]; {
        int n;
        for( n=0; s[n] != '\0'; )
                n++;
        return(n);
}
```

Rewriting with pointers gives

```
length(s)
   char *s; {
        int n;
        for( n=0; *s != '\0'; s++ )
                n++;
        return(n);
}
```

You can now see why we have to say what kind of thing **s** points to — if we're to increment it with s++ we have to increment it by the right amount.

The pointer version is more efficient (this is almost always true) but even more compact is

```
for( n=0; *s++ != '\0'; n++ );
```

The '*s' returns a character; the '++' increments the pointer so we'll get the next character next time around. As you can see, as we make things more efficient, we also make them less clear. But '*s++' is an idiom so common that you have to know it.

Going a step further, here's our function **strcopy** that copies a character array **s** to another **t**.

```
strcopy(s,t)
   char *s, *t; {
        while(*t++ = *s++);
}
```

We have omitted the test against '\0', because '\0' is identically zero; you will often see the code this way. (You *must* have a space after the '=': see section 25.)

For arguments to a function, and there only, the declarations

```
char s[ ];
char *s;
```

are equivalent — a pointer to a type, or an array of unspecified size of that type, are the same thing.

If this all seems mysterious, copy these forms until they become second nature. You don't often need anything more complicated.

17. Function Arguments

Look back at the function **strcopy** in the previous section. We passed it two string names as arguments, then proceeded to clobber both of them by incrementation. So how come we don't lose the original strings in the function that called **strcopy**?

As we said before, C is a "call by value" language: when you make a function call like f(x), the *value* of x is passed, not its address. So there's no way to *alter* x from inside f. If x is an array (**char** x[10]) this isn't a problem, because x *is* an address anyway, and you're not trying to change it, just what it addresses. This is why **strcopy** works as it does. And it's convenient not to have to worry about making temporary copies of the input arguments.

But what if x is a scalar and you do want to change it? In that case, you have to pass the *address* of x to f, and then use it as a pointer. Thus for example, to interchange two integers, we must write

```
flip(x, y)
   int *x, *y; {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

and to call flip, we have to pass the addresses of the variables:

```
flip (&a, &b);
```

18. Multiple Levels of Pointers; Program Arguments

When a C program is called, the arguments on the command line are made available to the main program as an argument count **argc** and an array of character strings **argv** containing the arguments. Manipulating these arguments is one of the most common uses of multiple levels of pointers ("pointer to pointer to ..."). By convention, **argc** is greater than zero; the first argument (in argv[0]) is the command name itself.

Here is a program that simply echoes its arguments.

```
main(argc, argv)
   int argc;
   char **argv; {
        int i;
        for( i = 1; i < argc; i++ )
                printf("%s ", argv[i]);
        putchar('\n');
}
```

Step by step: **main** is called with two arguments, the argument count and the array of arguments. **argv** is a pointer to an array, whose individual elements are pointers to arrays of char-

acters. The zeroth argument is the name of the command itself, so we start to print with the first argument, until we've printed them all. Each **argv[i]** is a character array, so we use a '**%s**' in the **printf**.

You will sometimes see the declaration of **argv** written as

```
char *argv[ ];
```

which is equivalent. But we can't use **char argv[ ][ ]**, because both dimensions are variable and there would be no way to figure out how big the array is.

Here's a bigger example using **argc** and **argv**. A common convention in C programs is that if the first argument is '−', it indicates a flag of some sort. For example, suppose we want a program to be callable as

```
prog −abc arg1 arg2 ...
```

where the '−' argument is optional; if it is present, it may be followed by any combination of a, b, and c.

```
main(argc, argv)
  int argc;
  char **argv; {
      ...
      aflag = bflag = cflag  = 0;
      if( argc > 1 && argv[1][0] == '−' ) {
            for( i=1; (c=argv[1][i]) != '\0'; i++ )
                  if( c=='a' )
                        aflag++;
                  else if( c=='b' )
                        bflag++;
                  else if( c=='c' )
                        cflag++;
                  else
                        printf("%c?\n", c);
            −−argc;
            ++argv;
      }
      ...
```

There are several things worth noticing about this code. First, there is a real need for the left-to-right evaluation that && provides; we don't want to look at **argv[1]** unless we know it's there. Second, the statements

```
−−argc;
++argv;
```

let us march along the argument list by one position, so we can skip over the flag argument as if it had never existed — the rest of the program is independent of whether or not there was a flag argument. This only works because **argv** is a pointer which can be incremented.

19. The Switch Statement; Break; Continue

The **switch** statement can be used to replace the multi-way test we used in the last example. When the tests are like this:

```
if( c == 'a' ) ...
else if( c == 'b' ) ...
else if( c == 'c' ) ...
else ...
```

testing a value against a series of *constants*, the switch statement is often clearer and usually gives better code. Use it like this:

```
switch( c ) {

case 'a':
        aflag++;
        break;
case 'b':
        bflag++;
        break;
case 'c':
        cflag++;
        break;
default:
        printf("%c?\n", c);
        break;
}
```

The **case** statements label the various actions we want; **default** gets done if none of the other cases are satisfied. (A **default** is optional; if it isn't there, and none of the cases match, you just fall out the bottom.)

The **break** statement in this example is new. It is there because the cases are just labels, and after you do one of them, you *fall through* to the next unless you take some explicit action to escape. This is a mixed blessing. On the positive side, you can have multiple cases on a single statement; we might want to allow both upper and lower case letters in our flag field, so we could say

```
case 'a': case 'A':     ...
case 'b': case 'B':     ...
 etc.
```

But what if we just want to get out after doing **case 'a'** ? We could get out of a **case** of the **switch** with a label and a **goto**, but this is really ugly. The **break** statement lets us exit without either **goto** or label.

```
switch( c ) {

case 'a':
        aflag++;
        break;
case 'b':
        bflag++;
        break;
...
}
/* the break statements get us here directly */
```

The **break** statement also works in **for** and **while** statements — it causes an immediate exit from the loop.

The **continue** statement works *only* inside **for**'s and **while**'s; it causes the next iteration of the loop to be started. This means it goes to the increment part of the **for** and the test part of the **while**. We could have used a **continue** in our example to get on with the next iteration of the **for**, but it seems clearer to use **break** instead.

20. Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example, if we were writing a compiler or assembler, we might need for each identifier information like its name (a character array), its source line number (an integer), some type information (a character, perhaps), and probably a usage count (another integer).

```
char    id[10];
int     line;
char    type;
int     usage;
```

We can make a structure out of this quite easily. We first tell C what the structure will look like, that is, what kinds of things it contains; after that we can actually reserve storage for it, either in the same statement or separately. The simplest thing is to define it and allocate storage all at once:

```
struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym;
```

This defines sym to be a structure with the specified shape; id, line, type and usage are *members* of the structure. The way we refer to any particular member of the structure is

```
structure-name . member
```

as in

```
sym.type = 077;
if( sym.usage == 0 ) ...
while( sym.id[j++] ) ...
  etc.
```

Although the names of structure members never stand alone, they still have to be unique — there can't be another id or usage in some other structure.

So far we haven't gained much. The advantages of structures start to come when we have arrays of structures, or when we want to pass complicated data layouts between functions. Suppose we wanted to make a symbol table for up to 100 identifiers. We could extend our definitions like

```
char    id[100][10];
int     line[100];
char    type[100];
int     usage[100];
```

but a structure lets us rearrange this spread-out information so all the data about a single identifer is collected into one lump:

```
struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100];
```

This makes sym an array of structures; each array element has the specified shape. Now we can refer to members as

```
sym[i].usage++;        /* increment usage of i-th identifier */
for( j=0; sym[i].id[j++] != '\0'; ) ...
   etc.
```

Thus to print a list of all identifiers that haven't been used, together with their line number,

```
for( i=0; i<nsym; i++ )
        if( sym[i].usage == 0 )
                printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Suppose we now want to write a function lookup(name) which will tell us if name already exists in sym, by giving its index, or that it doesn't, by returning a −1. We can't pass a structure to a function directly — we have to either define it externally, or pass a pointer to it. Let's try the first way first.

```
int     nsym  0;      /* current length of symbol table */
struct {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100];           /* symbol table */
main( ) {
        ...
        if( (index = lookup(newname)) >= 0 )
                sym[index].usage++;          /* already there ... */
        else
                install(newname, newline, newtype);
        ...
}
lookup(s)
   char *s; {
        int i;
        extern struct {
                char    id[10];
                int     line;
                char    type;
                int     usage;
        } sym[ ];
        for( i=0; i<nsym; i++ )
                if( compar(s, sym[i].id) > 0 )
                        return(i);
        return(-1);
}

compar(s1,s2)         /*  return 1 if s1==s2, 0 otherwise */
   char *s1, *s2; {
        while( *s1++ == *s2 )
                if( *s2++ == '\0' )
                        return(1);
```

```
        return(0);
}
```

The declaration of the structure in **lookup** isn't needed if the external definition precedes its use in the same source file, as we shall see in a moment.

Now what if we want to use pointers?

```
struct  symtag {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100], *psym;

        psym = &sym[0];       /* or psym = sym; */
```

This makes **psym** a pointer to our kind of structure (the symbol table), then initializes it to point to the first element of **sym.**

Notice that we added something after the word **struct**: a "tag" called **symtag.** This puts a name on our structure definition so we can refer to it later without repeating the definition. It's not necessary but useful. In fact we could have said

```
struct  symtag {
        ... structure definition
};
```

which wouldn't have assigned any storage at all, and then said

```
struct  symtag sym[100];
struct  symtag *psym;
```

which would define the array and the pointer. This could be condensed further, to

```
struct  symtag sym[100], *psym;
```

The way we actually refer to an member of a structure by a pointer is like this:

```
        ptr -> structure-member
```

The symbol '->' means we're pointing at a member of a structure; '->' is only used in that context. **ptr** is a pointer to the (base of) a structure that contains the structure member. The expression **ptr->structure-member** refers to the indicated member of the pointed-to structure. Thus we have constructions like:

```
psym->type = 1;
psym->id[0] = 'a';
```

and so on.

For more complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example,

```
struct { int x, *y; } *p;
p->x++      increments x
++p->x      so does this!
(++p)->x    increments p before getting x
*p->y++     uses y as a pointer, then increments it
*(p->y)++   so does this
*(p++)->y   uses y as a pointer, then increments p
```

The way to remember these is that ->, . (dot), ( ) and [ ] bind very tightly. An expression in-

volving one of these is treated as a unit. p->x, a[i], y.x and f(b) are names exactly as abc is.

If p is a pointer to a structure, any arithmetic on p takes into account the acutal size of the structure. For instance, p++ increments p by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes of its members — because of alignments of different sized objects, there may be "holes" in a structure.

Enough theory. Here is the lookup example, this time with pointers.

```
struct symtag {
        char    id[10];
        int     line;
        char    type;
        int     usage;
} sym[100];

main( ) {
        struct symtag *lookup( );
        struct symtag *psym;
        ...
        if( (psym = lookup(newname)) )      /* non-zero pointer */
                psym -> usage++;            /* means already there */
        else
                install(newname, newline, newtype);
        ...
}

struct symtag *lookup(s)
  char *s; {
        struct symtag *p;
        for( p=sym; p < &sym[nsym]; p++ )
                if( compar(s, p->id) > 0)
                        return(p);
        return(0);
}
```

The function compar doesn't change: 'p->id' refers to a string.

In main we test the pointer returned by lookup against zero, relying on the fact that a pointer is by definition never zero when it really points at something. The other pointer manipulations are trivial.

The only complexity is the set of lines like

```
struct symtag *lookup( );
```

This brings us to an area that we will treat only hurriedly — the question of function types. So far, all of our functions have returned integers (or characters, which are much the same). What do we do when the function returns something else, like a pointer to a structure? The rule is that any function that doesn't return an int has to say explicitly what it does return. The type information goes before the function name (which can make the name hard to see). Examples:

```
char f(a)
  int a; {
        ...
}
```

```
int *g( ) { ... }

struct symtag *lookup(s) char *s; { ... }
```

The function f returns a character, g returns a pointer to an integer, and lookup returns a pointer to a structure that looks like **symtag**. And if we're going to use one of these functions, we have to make a declaration where we use it, as we did in **main** above.

Notice the parallelism between the declarations

```
struct symtag *lookup( );
struct symtag *psym;
```

In effect, this says that lookup( ) and psym are both used the same way — as a pointer to a strcture — even though one is a variable and the other is a function.

21. Initialization of Variables

An external variable may be initialized at compile time by following its name with an initializing value when it is defined. The initializing value has to be something whose value is known at compile time, like a constant.

```
int     x       0;      /* "0" could be any constant */
int     a       'a';
char    flag    0177;
int     *p      &y[1];  /* p now points to y[1] */
```

An external array can be initialized by following its name with a list of initializations enclosed in braces:

```
int     x[4]    {0,1,2,3};              /* makes x[i] = i */
int     y[ ]    {0,1,2,3};              /* makes y big enough for 4 values */
char    *msg    "syntax error\n";       /* braces unnecessary here */
char *keyword[ ]{
        "if",
        "else",
        "for",
        "while",
        "break",
        "continue",
        0
};
```

This last one is very useful — it makes keyword an array of pointers to character strings, with a zero at the end so we can identify the last element easily. A simple lookup routine could scan this until it either finds a match or encounters a zero keyword pointer:

```
lookup(str)             /* search for str in keyword[ ] */
   char *str; {
        int i,j,r;
        for( i=0; keyword[i] != 0; i++) {
                for( j=0; (r=keyword[i][j]) == str[j] && r != '\0'; j++ );
                if( r == str[j] )
                        return(i);
        }
        return(-1);
}
```

Sorry — neither local variables nor structures can be initialized.

22. Scope Rules: Who Knows About What

A complete C program need not be compiled all at once; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. How do we arrange that data gets passed from one routine to another? We have already seen how to use function arguments and values, so let us talk about external data. Warning: the words *declaration* and *definition* are used precisely in this section; don't treat them as the same thing.

A major shortcut exists for making **extern** declarations. If the definition of a variable appears *before* its use in some function, no **extern** declaration is needed within the function. Thus, if a file contains

```
f1( ) { ... }
int foo;
f2( ) { ... foo = 1; ... }
f3( ) { ... if ( foo ) ... }
```

no declaration of foo is needed in either f2 or or f3, because the external definition of foo appears before them. But if f1 wants to use foo, it has to contain the declaration

```
f1( ) {
        extern int foo;
        ...
}
```

This is true also of any function that exists on another file — if it wants foo it has to use an **extern** declaration for it. (If somewhere there is an **extern** declaration for something, there must also eventually be an external definition of it, or you'll get an "undefined symbol" message.)

There are some hidden pitfalls in external declarations and definitions if you use multiple source files. To avoid them, first, define and initialize each external variable only once in the entire set of files:

```
int     foo     0;
```

You can get away with multiple external definitions on UNIX, but not on GCOS, so don't ask for trouble. Multiple initializations are illegal everywhere. Second, at the beginning of any file that contains functions needing a variable whose definition is in some other file, put in an **extern** declaration, outside of any function:

```
extern int      foo;
f1( ) { ... }
  etc.
```

The **#include** compiler control line, to be discussed shortly, lets you make a single copy of the external declarations for a program and then stick them into each of the source files making up the program.

23. #define, #include

C provides a very limited macro facility. You can say

```
#define         name            something
```

and thereafter anywhere "name" appears as a token, "something" will be substituted. This is

particularly useful in parameterizing the sizes of arrays:

```
#define     ARRAYSIZE   100
    int     arr[ARRAYSIZE];
    ...
    while( i++ < ARRAYSIZE )...
```

(now we can alter the entire program by changing only the **define**) or in setting up mysterious constants:

```
#define     SET         01
#define     INTERRUPT   02      /* interrupt bit */
#define     ENABLED     04
...
if( x & (SET | INTERRUPT | ENABLED) ) ...
```

Now we have meaningful words instead of mysterious constants. (The mysterious operators '&' (AND) and '|' (OR) will be covered in the next section.) It's an excellent practice to write programs without any literal constants except in **#define** statements.

There are several warnings about **#define.** First, there's no semicolon at the end of a **#define**; all the text from the name to the end of the line (except for comments) is taken to be the "something". When it's put into the text, blanks are placed around it. Good style typically makes the name in the **#define** upper case — this makes parameters more visible. Definitions affect things only after they occur, and only within the file in which they occur. Defines can't be nested. Last, if there is a **#define** in a file, then the first character of the file *must* be a '#', to signal the preprocessor that definitions exist.

The other control word known to C is **#include.** To include one file in your source at compilation time, say

```
#include "filename"
```

This is useful for putting a lot of heavily used data definitions and **#define** statements at the beginning of a file to be compiled. As with **#define**, the first line of a file containing a **#include** has to begin with a '#'. And **#include** can't be nested — an included file can't contain another **#include.**

24. Bit Operators

C has several operators for logical bit-operations. For example,

```
x = x & 0177;
```

forms the bit-wise AND of x and 0177, effectively retaining only the last seven bits of **x.** Other operators are

|	inclusive OR
^	(circumflex) exclusive OR
~	(tilde) 1's complement
!	logical NOT
<<	left shift (as in x<<2)
>>	right shift (arithmetic on PDP-11; logical on H6070, IBM360)

25. Assignment Operators

An unusual feature of C is that the normal binary operators like '+', '-', etc. can be combined with the assignment operator '=' to form new assignment operators. For example,

```
x =- 10;
```

uses the assignment operator '=-' to decrement x by 10, and

```
x =& 0177
```

forms the AND of x and 0177. This convention is a useful notational shortcut, particularly if x is a complicated expression. The classic example is summing an array:

```
for( sum=i=0; i<n; i++ )
        sum =+ array[i];
```

But the spaces around the operator are critical! For instance,

```
x = -10;
```

sets x to -10, while

```
x =- 10;
```

subtracts 10 from x. When no space is present,

```
x=-10;
```

also decreases x by 10. This is quite contrary to the experience of most programmers. In particular, watch out for things like

```
c=*s++;
y=&x[0];
```

both of which are almost certainly not what you wanted. Newer versions of various compilers are courteous enough to warn you about the ambiguity.

Because all other operators in an expression are evaluated before the assignment operator, the order of evaluation should be watched carefully:

```
x = x<<y | z;
```

means "shift x left y places, then OR with z, and store in x." But

```
x =<< y | z;
```

means "shift x left by y|z places", which is rather different.

26. Floating Point

We've skipped over floating point so far, and the treatment here will be hasty. C has single and double precision numbers (where the precision depends on the machine at hand). For example,

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
        sum =+ y[i];
avg = sum/n;
```

forms the sum and average of the array y.

All floating arithmetic is done in double precision. Mixed mode arithmetic is legal; if an arithmetic operator in an expression has both operands **int** or **char**, the arithmetic done is integer, but if one operand is **int** or **char** and the other is **float** or **double**, both operands are con-

verted to **double**. Thus if i and j are **int** and x is **float**,

```
(x+i)/j      converts i and j to float
x + i/j      does i/j integer, then converts
```

Type conversion may be made by assignment; for instance,

```
int m, n;
float x, y;
m = x;
y = n;
```

converts x to integer (truncating toward zero), and n to floating point.

Floating constants are just like those in Fortran or PL/I, except that the exponent letter is 'e' instead of 'E'. Thus

```
pi = 3.14159;
large = 1.23456789e10;
```

printf will format floating point numbers: "%w.df" in the format string will print the corresponding variable in a field w digits wide, with d decimal places. An e instead of an f will produce exponential notation.

27. Horrors! goto's and labels

C has a **goto** statement and labels, so you can branch about the way you used to. But most of the time **goto**'s aren't needed. (How many have we used up to this point?) The code can almost always be more clearly expressed by **for/while**, **if/else**, and compound statements.

One use of **goto**'s with some legitimacy is in a program which contains a long loop, where a **while(1)** would be too extended. Then you might write

```
mainloop:
    ...
    goto mainloop;
```

Another use is to implement a **break** out of more than one level of **for** or **while**. **goto**'s can only branch to labels within the same function.

28. Acknowledgements

I am indebted to a veritable host of readers who made valuable criticisms on several drafts of this tutorial. They ranged in experience from complete beginners through several implementors of C compilers to the C language designer himself. Needless to say, this is a wide enough spectrum of opinion that no one is satisfied (including me); comments and suggestions are still welcome, so that some future version might be improved.

References

C is an extension of B, which was designed by D. M. Ritchie and K. L. Thompson [4]. The C language design and UNIX implementation are the work of D. M. Ritchie. The GCOS version was begun by A. Snyder and B. A. Barres, and completed by S. C. Johnson and M. E. Lesk. The IBM version is primarily due to T. G. Peterson, with the assistance of M. E. Lesk.

[1] D. M. Ritchie, *C Reference Manual.* Bell Labs, Jan. 1974.

[2] M. E. Lesk & B. A. Barres, *The GCOS C Library.* Bell Labs, Jan. 1974.

[3] D. M. Ritchie & K. Thompson, *UNIX Programmer's Manual.* 5th Edition, Bell Labs, 1974.

[4] S. C. Johnson & B. W. Kernighan, *The Programming Language B.* Computer Science Technical Report 8, Bell Labs, 1972.

A New Input-Output Package

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

A new package of IO routines is available. It was designed with the following goals in mind.

1. It should be similar in spirit to the earlier Portable Library, and, to the extent possible, be compatible with it. At the same time a few dubious design choices in the Portable Library will be corrected.
2. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
3. It must be simple to use, and also free of the magic numbers and mysterious calls the use of which mars the understandability and portability of many programs using older packages.
4. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP11 running a version of Unix.

It is intended that this package replace the Portable Library. Although it is not directly compatible, as discussed below, it is sufficiently similar that modifying programs to use it should be a simple exercise.

The most crucial difference between this package and the Portable Library is that the current offering names streams in terms of pointers rather than by the integers known as 'file descriptors.' Thus, for example, the routine which opens a named file returns a pointer to a certain structure rather than a number; the routine which reads an open file takes as an argument the pointer returned from the open call.

General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The library containing the routines is '/usr/lib/libS.a,' so the command to compile is

```
cc ... -lS
```

All names in the include file intended only for internal use begin with an underscore '_' to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin	The name of the standard input file
stdout	The name of the standard output file
stderr	The name of the standard error file
EOF	is actually −1, and is the value returned by the read routines on end-of-file or error.

NULL is a notation for the null pointer, returned by pointer-valued functions to indicate an error

FILE expands to 'struct _iob' and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an IO buffer supplied by the user. See *setbuf*, below.

getc, getchar, putc, putchar, feof, ferror, fileno
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package, like the Portable Library, offer the convenience of automatic buffer allocation and output flushing where appropriate. Absent, however, is the facility of changing the default input and output streams by assigning to 'cin' and 'cout.' The names 'stdin,' stdout,' and 'stderr' are in effect constants and may not be assigned to.

Calls

The routines in the library are in nearly one-to-one correspondence with those in the Portable Library. In several cases the name has been changed. This is an attempt to reduce confusion.

*FILE *fopen(filename, type) char *filename, *type*

Fopen opens the file and, if needed, allocates a buffer for it. *Filename* is a character string specifying the name. *Type* is a character string (not a single character). It may be '"r",' '"w",' or '"a"' to indicate intent to read, write, or append. The value returned is a file pointer. If it is NULL the attempt to open failed.

*FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr*

The stream named by *ioptr* is closed, if necessary, and then reopened as if by *fopen*. If the attempt to open fails, NULL is returned, otherwise *ioptr*, which will now refer to the new file. Often the reopened stream is *stdin* or *stdout*.

*int getc(ioptr) FILE *ioptr*

returns the next character from the stream named by *ioptr*, which is a pointer to a file such as returned by *fopen*, or the name *stdin*. The integer EOF is returned on end-of-file or when an error occurs. The null character '\0' is a legal character.

*int fgetc(ioptr) FILE *ioptr*

acts like *getc* but is a genuine function, not a macro.

*putc(c, ioptr) FILE *ioptr*

Putc writes the character *c* on the output stream named by *ioptr*, which is a value returned from *fopen* or perhaps *stdout* or *stderr*. The character is returned as value, but EOF is returned on error.

*fputc(c, ioptr) FILE *ioptr*

Fputc acts like *putc* but is a genuine function, not a macro.

*fclose(ioptr) FILE *ioptr*

The file corresponding to *ioptr* is closed after any buffers are emptied. A buffer allocated by the IO system is freed. *Fclose* is automatic on normal termination of the program.

*fflush(ioptr) FILE *ioptr*

Any buffered information on the (output) stream named by *ioptr* is written out. Output files are normally buffered if and only if they are not directed to the terminal, but *stderr* is unbuffered unless *setbuf* is used.

exit(errcode)

Exit terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls *fflush* for each output file. To terminate without flushing, use *_exit.*

*feof(ioptr) FILE *ioptr*

returns non-zero when end-of-file has occurred on the specified input stream.

*ferror(ioptr) FILE *ioptr*

returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar( )

is identical to *getc(stdin).*

putchar(c)

is identical to *putc(c, stdout).*

*char *gets(s) char *s*

reads characters up to a new-line from the standard input. The new-line character is replaced by a null character. It is the user's responsibility to make sure that the character array *s* is large enough. *Gets* returns its argument, or NULL if end-of-file or error occurred. Note that this routine is not compatible with *fgets;* it is included for downward compatibility.

*char *fgets(s, n, ioptr) char *s; FILE *ioptr*

reads up to *n* characters from the stream *ioptr* into the character pointer *s.* The read terminates with a new-line character. The new-line character is placed in the buffer followed by a null character. The first argument, or NULL if error or end-of-file occurred, is returned.

*puts(s) char *s*

writes the null-terminated string (character array) *s* on the standard output. A new-line is appended. No value is returned. Note that this routine is not compatible with *fputs;* it is included for downward compatibility.

*fputs(s, ioptr) char *s; FILE *ioptr*

writes the null-terminated string (character array) *s* on the stream *ioptr.* No new-line is appended. No value is returned.

*ungetc(c, ioptr) FILE *ioptr*

The argument character *c* is pushed back on the input stream named by *ioptr.* Only one character may be pushed back.

*printf(format, a1, . . .) char *format*

*fprintf(ioptr, format, a1, . . .) FILE *ioptr; char *format*

*sprintf(s, format, a1, . . .)char *s, *format*

Printf writes on the standard output. *Fprintf* writes on the named output stream. *Sprintf* puts characters in the character array (string) named by *s.* The specifications are as described in section *printf* (III) of the Unix Programmer's Manual. There is a new conversion: *%m.n*g converts a double argument in the style of *e* or *f* as most appropriate.

*scanf(format, a1, . . .) char *format*

*fscanf(ioptr, format, a1, . . .) FILE *ioptr; char *format*

*sscanf(s, format, a1, . . .) char *s, *format*

Scanf reads from the standard input. *Fscanf* reads from the named input stream. *Sscanf* reads from the character string supplied as *s.* The specifications are identical to those of the Portable Library. *Scanf* reads characters, interprets them according to a format, and stores the results in its arguments. It expects as arguments a control string *format,* described below, and a set of

arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which are ignored.
2. Ordinary characters (not %) which are expected to match the next non-space character of the input stream (where space characters are defined as blank, tab or newline).
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character , an optional numerical maximum field width, and a conversion character.

A conversion specification is used to direct the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by the character. An input field is defined as a string of non-space characters; it extends either to the next space character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

% indicates that a single % character is expected in the input stream at this point; no assignment is done.

d indicates that a decimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

o indicates that an octal integer is expected in the input stream; the corresponding argument should be a integer pointer.

x indicates that a hexadecimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

s indicates that a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

c indicates that a character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over space characters is suppressed in this case; to read the next non-space character, try *%1s.* If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

e (or *f*) indicates that a floating point number is expected in the input stream; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float.* The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field beginning with an E or e followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters *d, o* and *x* may be capitalized or preceded by *l* to indicate that a pointer to *long* rather than *int* is expected. Similarly, the conversion characters *e* or *f* may be capitalized or preceded by *l* to indicate that a pointer to *double* rather than *float* is in the argument list. The character *h* will function similarly in the future to indicate *short* data items.

For example, the call

```
int i; float x; char name[50];
scanf( "%d%f%s", &i, &x, name);
```

with the input line

```
25  54.32E-1  thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain *"thompson\0"*. Or,

```
int i; float x; char name[50];
scanf("%2d%f%d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip "0123", and place the string "56\0" in *name*. The next call to *getchar* will return 'a'.

Scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

*fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr*

reads *nitems* of data beginning at *ptr* from file *ioptr*. It behaves identically to the Portable Library's *cread*. No advance notification that binary IO is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the fopen call.

*fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr*

Like *fread*, but in the other direction.

*rewind(ioptr) FILE *ioptr*

rewinds the stream named by *ioptr*. It is not very useful except on input, since a rewound output file is still open only for output.

*system(string) char *string*

The *string* is executed by the shell as if typed at the terminal.

*getw(ioptr) FILE *ioptr*

returns the next word from the input stream named by *ioptr*. EOF is returned on end-of-file or error, but since this a perfectly good integer *feof* and *ferror* should be used.

*putw(w, ioptr) FILE *ioptr*

writes the integer *w* on the named output stream.

*setbuf(ioptr, buf) FILE *ioptr; char *buf*

Setbuf may be used after a stream has been opened but before IO has started. If *buf* is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It is a character array of sufficient size:

```
char    buf[BUFSIZ];
```

*fileno(ioptr) FILE *ioptr*

returns the integer file descriptor associated with the file.

*fseek(ioptr, offset, ptrname) FILE *ioptr; long offset*

The location of the next byte in the stream named by *ioptr* is adjusted. *Offset* is a long integer. If *ptrname* is 0, the offset is measured from the beginning of the file; if *ptrname* is 1, the offset

is measured from the current read or write pointer; if *ptrname* is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-Unix systems, the offset must be a value returned from *ftell* and the ptrname must be 0).

*long ftell(ioptr) FILE *ioptr*

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-Unix systems the value of this call is useful only for handing to *fseek*, so as to position the file to the same place it was when *ftell* was called.)

*getpw(uid, buf) char *buf*

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array *buf*, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

*char *calloc(num, size)*

allocates space for *num* items each of size *size*. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available.

*cfree(ptr) char *ptr*

Space is returned to the pool used by *calloc*. Disorder can be expected if the pointer was not obtained from *calloc*.

The following are macros defined by stdio.h.

isalpha(c)

returns non-zero if the argument is alphabetic.

isupper(c)

returns non-zero if the argument is upper-case alphabetic.

islower(c)

returns non-zero if the argument is lower-case alphabetic.

isdigit(c)

returns non-zero if the argument is a digit.

isspace(c)

returns non-zero if the argument is a spacing character: tab, new-line, carriage return, vertical tab, form feed, space.

toupper(c)

returns the upper-case character corresponding to the lower-case letter c.

tolower(c)

returns the lower-case character corresponding to the upper-case letter c.

A General-Purpose Subroutine Library for PWB/UNIX

Alan L. Glasser

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

A new general-purpose subroutine library has been written for PWB/UNIX. It complements the functions provided by D. M. Ritchie's *A New Input-Output Package.* This library has been used in the implementation of Release 4 of the Programmer's Workbench Source Code Control System (SCCS/PWB), and many small UNIX C programs. It is efficient in time and space, as well as being easy to use. This document is a user's guide to this library.

1. "Include" FILES

The directory "/usr/include" contains public *include* files. Users of the subroutine library should be familiar with the contents of these files. The following are available:

archive.h	Defines the "magic" number of an archive file and declares a structure for the header of an archive file.
ctype.h	Defines macros for testing whether a character is alphabetic, upper-case, lower-case, a digit, a "space," and for converting upper-case characters to lower-case and vice-versa.
dir.h	Declares a structure for a directory entry.
errnos.h	Defines system call error numbers (see *INTRO*(II)). These were copied from the UNIX system source.
fatal.h	Defines certain macros, constants, and variables used by the general-purpose error and signal handling subroutines (see below).
macros.h	Defines some general-purpose macros.
misc.h	Declares some unnamed structures for accessing pieces of a variable (e.g., the low byte of an integer). These were copied from the UNIX system source.
stat.h	Declares an inode structure to be used with *stat*(II) and *fstat*(II), and defines constants for the various mode bits of an inode.
stdio.h	Used by *A New Input-Output Package.*
time.h	Declares a structure to be used with *localtime*(III).
system.h	Defines certain system constants; in particular, signal numbers. These were copied from the UNIX system source.

2. SUBROUTINES

There are four sets of subroutines available. Three of these sets are kept in one library ("/usr/lib/libpw.a", accessible as *−lpw*), and the fourth set is kept in a second library ("/usr/lib/libwrt.a", accessible as *−lwrt*) for reasons that will be explained below; *−lwrt* should normally be last on the *cc* argument list.

The first library (*−lpw*) contains the *string, error,* and *sys* sets. The second library (*−lwrt*) contains only the *write* set (see below).

The *string* set can be used in conjunction with any other subroutines. The *string* set is independent of the other sets and of other subroutine libraries; no subroutine in the *string* set calls any subroutine not in the *string* set.

The *error* set provides general-purpose error and signal handling routines.

The *sys* set provides interfaces to most of the commonly used system calls. These interfaces will call *fatal()* (see the *error* set) if an error condition is detected in the interface.

The *write* set contains an interface to the *write*(II) system call; this interface handles errors and calls *fatal()*, if necessary. If the *write* set is used, then all calls to the subroutine *write()* will be directed to this write routine (remember, it will call *fatal()* if an error is detected). It is for this reason that the *write* set is in a separate library; if all routines were in the same library, then users would be unwittingly using the *write*(II) routine of the *write* set.

The subroutines presented here do not call *nargs*(III), as the *nargs*(III) subroutine does not work if a program is loaded with separate data and instruction spaces.

2.1 String Set

*char *alloca(nbytes)*

Allocates *nbytes* bytes of automatic memory. Automatic memory is freed upon return from the calling function (space is allocated from the stack). There is no way to *explicitly* free a piece of memory gotten from *alloca()*. *Alloca()* returns the address of the allocated area; a memory fault is generated when there isn't enough memory.

N. B.: Use of *alloca()* as an argument to another subroutine will *not* work correctly because arguments are pushed onto the stack and *alloca()* takes memory from the stack. It is necessary to first set a temporary variable equal to the value returned by *alloca()*, and then pass that variable to the other subroutine.

any(c, str)

If character *c* is equal to any character in the string *str*, returns 1; else returns 0.

anystr(str1, str2)

If any character of string *str1* is equal to any character in string *str2*, returns the offset (in *str1*) of the first such match; else returns −1.

balbrk(str, open, clos, end)

Finds the offset, in string *str*, of the first of the characters in the string *end* occurring *outside* of a *balanced string*. A balanced string contains matched occurrences of any character in the string *open* and the corresponding character in the string *clos*. Balanced strings may be nested. In addition to the characters in *end*, the null character is implicitly an *end* character. Unmatched members of *open* or *close* result in an error return (a value of −1 is returned).

Example 1:

```
s = "a[bc=2]=3";
o = "([[";
c = ")]]";
e = "=";
```

balbrk(s, o, c, e) returns 7.

Example 2:

```
s = "a[bc=2=3";
```

with o, c, and e as in Example 1, balbrk(s, o, c, e) returns −1.

*char *cat(dest, $source_1$, $source_2$, $source_3$, ..., $source_n$, 0)*

Concatenates strings. First, string $source_1$ is copied to string *dest*. Then subsequent $source_k$ strings are concatenated (by copying) onto the end of *dest*. The space for *dest* must be allocated by the caller (i.e., *dest* is taken to be the address of an area of memory large enough to hold the result). The address of the result (i.e., *dest*) is returned.

dname(pathname)
Returns a pointer to the name of the directory that contains the file pointed to by *pathname. Dname()* is the complement of *sname()* (see below). If *pathname* is a simple name (e.g., "file"), a pointer to "." is returned. If *pathname* is "/unix", a pointer to "/" is returned. If *pathname* is "/bin/who", a pointer to "/bin" is returned, etc. The string pointed to by *pathname* is modified by *dname(); pathname* is returned.

equal(str1, str2)
If string *str1* is equal to string *str2*, returns 1; else returns 0.

imatch(prefix, str)
Initial match. If string *prefix* is a prefix of string *str*, returns 1; else returns 0.

index(str1, str2)
If string *str2* is a substring of string *str1*, returns the offset of the first occurrence of *str2* in *str1*, else returns −1.

move(a, b, n)
Copies the first *n* characters from string *a* to string *b*.

patoi(str)
Converts an ASCII string to a integer. The string *str* is taken to be a string of decimal digits; the numeric value represented by *str* is returned. Converts positive numbers only. Returns −1 if a non-numeric character is encountered.

long patol(str)
Converts an ASCII string to a long integer. The string *str* is taken to be a string of decimal digits; the numeric value represented by *str* is returned. Converts positive numbers only. Returns −1 if a non-numeric character is encountered.

*char *repeat(result, str, repfac)*
The string *str* is first copied to the string *result.* Then *str* is copied *repfac−1* times onto the end of *result.* As with *cat()* (see above), allocation of space for *result* is the caller's responsibility. *Result* is returned.

*char *satoi(str, ip)*
Satoi() is similar to *patoi* (see above), except that the integer value is stored through the integer pointer *ip*, and a pointer to the first non-numeric character encountered is returned.

size(str)
Returns the number of bytes of memory used by string *str*, including the null byte, so that *size(str)* is equal to *length(str)+1.*

*char *sname(str)*
Sname() returns a pointer to the "simple" name of path name *str;* i.e., a pointer to the first character after the last "/" in *str.* If *str* does not contain a "/", a pointer to the original string is returned.

*char *strend(str)*
Strend() returns a pointer to the end (null byte) of the string *str.*

substr(str, result, origin, len)
Copies at most *len* characters from the string *str* starting at *str[origin]* to the string pointed to by *result.* Sufficient space must exist for that string; *result* is returned. There is *no checking* for the reasonableness of the arguments. The copying of *str* to *result* stops if either the specified number (i.e., *len*, which is taken as an unsigned integer) characters have been copied, or if the end of *str* (i.e., a null byte) is found. A large value of *len* (e.g., −1) will usually cause all of *str* to be copied.

*char *trnslat(str, old, new, result)*
Copies string *str* to string *result* replacing any character found in string *old* with the corresponding character from string *new; result* is returned.

verify(str1, str2)
If string *str1* contains any characters not in string *str2*, returns the offset of the first such character in *str1*; else returns −1.

*char *zero(ptr, cnt)*
Sets to zero the area of memory *cnt* bytes long, starting at address *ptr*; *ptr* is returned.

*char *zeropad(str)*
Replace initial blanks with "0" characters in string *str*; *str* is returned.

2.2 Error Set

The *error* set of subroutines consists of a general-purpose error handling routine called *fatal()*, and general-purpose signal-setting and signal-catching routines called *setsig()* and *setsig1()*, respectively. There are also two additional routines called *clean_up()* and *userexit()*, which may be called by *fatal()* or *setsig1()*. Default versions of these two additional routines are supplied in the library. Users may define their own *clean_up()* and *userexit()* routines.

The public *include* file "fatal.h" contains definitions needed to use *fatal()*. It contains the following:

```
extern int      Fflags;
extern char     *Ffile;
extern int      Fvalue;
extern int      (*Ffunc)();
extern int      Fjmp[3];

# define FTLMSG     0100000
# define FTLCLN      040000
# define FTLFUNC     020000
# define FTLACT         077
# define FTLJMP          02
# define FTLEXIT         01
# define FTLRET           0

# define FSAVE(val) SAVE(Fflags,old_Fflags); Fflags = val;
# define FRSTR()    RSTR(Fflags,old_Fflags);
```

fatal(msg)
A general-purpose error handler. Typically, low-level subroutines that detect error conditions (an open or create routine, for example) return as a value a call of *fatal()* with an appropriate message string. For example:

```
return(fatal("can't do it"));
```

Higher-level routines control the execution of *fatal()* via the global word *Fflags*. The macros FSAVE() and FRSTR() in "fatal.h" can be used by higher-level subroutines to save and restore the *Fflags* word.

The argument to *fatal()* is a pointer to a message string. The action of *fatal()* is driven completely from the *Fflags* global integer, which is interpreted as explained below.

The FTLMSG bit controls the writing of the message on file descriptor 2. The message is preceded by the string "ERROR: ", unless the global character pointer *Ffile* is non-zero, in which case the message is preceded by a string equivalent to:

```
s = sprintf(space, "ERROR [%s]: ", Ffile);
```

A new-line character is written after the user-supplied message.

If the FTLCLN bit is on, *clean_up()* is called with an argument of 0 (see below).

If the FTLFUNC bit is on, the function pointed to by the global function pointer *Ffunc* is called with the user-supplied message pointer as an argument. This feature can be used to log these messages.

The FTLACT bits determine how *fatal()* should return. If the FTLJMP bit is one, *longjmp(Fjmp)* (see *setjmp*(III)) is called. If the FTLEXIT bit is one the value of *userexit(1)* is passed as an argument to

exit(II) (see below). If none of the FTLACT bits is on (the default value for *Fflags* is 0), the global word *Fvalue* (initialized to −1) is returned.

If all *fatal()* globals have their default values, *fatal()* simply returns −1.

setsig()

General-purpose signal-setting routine. All signals not already ignored or caught are made to be caught by the signal catching routine *setsig1()*.

setsig1()

General-purpose signal catcher and termination routine. If a signal other than hangup, interrupt, or quit is caught, a "user-oriented" *help*(I) message is printed on file descriptor 2. If hangup, interrupt, or quit is caught, subsequent occurrences of that signal will be ignored. Termination is similar to the FTLCLN and FTLEXIT options of *fatal()*, in that *clean_up(sig)* (where *sig* is the signal number) and *exit(userexit(1))* are called.

If the file "dump.core" exists in the current directory, the IOT signal is set to 0 and *abort*(III) is called to produce a core dump (after calling *clean_up()*, but before calling *userexit()*).

clean_up()

A default *clean_up()* routine is provided to resolve external references. It simply returns. User-supplied *clean_up()* routines are often used for removing temporary files, etc.

userexit(code)

A default *userexit()* routine is provided to resolve external references. It returns the value of *code*. User-supplied *userexit()* routines are often used for logging usage statistics.

2.3 Sys Set

The *sys* set of subroutines provides interfaces to system calls that process error conditions and call *fatal()*. In addition, a few functions which are not available elsewhere are provided.

curdir(path)

Places the complete pathname of the current directory in string *path*. Returns 0 on success, non-zero on failure. On successful return, the current directory is the same as it was on entry; on failure return, the current directory is not known.

fdfopen(fd, mode)

This subroutine provides a file-descriptor interface to the routines in *A New Input-Output Package*, and is required when one wants to use the routines in *A New Input-Output Package* with pipes. The first argument is a file descriptor (from *open*(II), *creat*(II), or *pipe*(II)), the second is the read/write mode (0/1, respectively). A *file pointer* (see *A New Input-Output Package*) is returned on success, and NULL on failure (typically, because there are no file structures available).

giveup(dump)

This routine does the following:

Change directory to "/" if argument is 0.
Set IOT signal to system default (0).
Call *abort*(III).

Thus, if *giveup()* is called with a 0 argument, and the file "/core" is not writable (or if the file "/core" doesn't exist, and the directory "/" is not writable), no core dump will be produced.

lockit(lockfile, count, pid)

A process semaphore implemented with files; typically, used to establish exclusive use of a resource (usually a file). The file's name is *lockfile*. *Lockit()* tries *count* times to create *lockfile* mode 444. It sleeps 10 seconds between tries. If *lockfile* is created, the number *pid* (typically, the process ID of the current process) is written (in binary; i.e., as two bytes) into *lockfile*, and 0 is returned. If *lockfile* exists and hasn't been modified within the last 60 seconds, and if it either is empty or if its first two bytes, interpreted as a binary number, are *not* the process ID of any existing process, *lockfile* is removed and *lockit()* tries again to make *lockfile*. After *count* tries, or if the reason for the creation of *lockfile* failing is something other than EACCES (see *INTRO*(II)), *lockit()* returns −1. See also *unlockit()*, below.

rename(oldname, newname)
Renames *oldname* to be *newname;* it can be thought of as:

```
mv oldname newname
```

It calls *xlink()* and *xunlink()* (see below).

unlockit(lockfile, pid)
Unlockit() is meant to be used to remove a *lockfile* created by *lockit()*. It verifies that the *pid* specified is contained in the first two bytes of the named *lockfile*, and then removes it. If the *pids* match, and the file is successfully removed, *unlockit()* returns 0; otherwise, −1 is returned.

userdir(uid)
Returns user's login directory name. The argument must be an integer user ID. There is an assumption that the directory field is the fifth field of a password file entry (i.e., there is no "group id" in the password file). Returns a pointer to the login directory on success, 0 on failure. It remembers its argument and the returned login directory name for subsequent calls to speed itself up. Users of PWB/UNIX systems should use *logdir()* (see *loginfo*(II)).

username(uid)
Returns user's login name. The argument must be an integer user ID. Returns a pointer to the login name on success, a pointer to the string representation of the user ID on failure. There is an assumption that the login name field is the first field of a password file entry. It remembers its argument and the returned login name for subsequent calls to speed itself up. Users of PWB/UNIX systems should use *logname()* (see *loginfo*(II)).

xalloc(size), xfree(ptr), xfreeall()
Xalloc() and *xfree()* are used in the same way as *alloc*(III) and *free*(III). The function *xfreeall()* frees all memory allocated by *xalloc()* (it calls *brk*(II)). *Xalloc()* returns the address of the allocated area on success, and the value of *fatal()* on failure. *Xfree()* and *xfreeall()* don't return anything. *Xalloc()* uses a "first fit" strategy (unlike *alloc*(II)). *Xfree()* always coalesces contiguous free blocks. *Xalloc()* always allocates 2-byte words. *Xalloc()* actually allocates one more word than the amount requested. The extra word (the first word of the allocated block) contains the size (in bytes) of the entire block. This size is used by *xfree()* to identify contiguous blocks, and is used by *xalloc()* to implement the first fit strategy. Bad things will happen if that first (size) word is overwritten. Worse things happen if *xfree()* is called with a garbage argument.

xcreat(name, mode)
Xcreat() is used in the same way as *creat*(II). *Xcreat()* requires write permission in the pertinent directory in *all* cases, and the created file is guaranteed to have the specified *mode* and be owned by the effective user (*xcreat()* does this by first unlinking the file to be created); *xcreat()* returns a file descriptor on success, and the value of *fatal()* on failure.

xfcreat(file, mode)
Xfcreat() is a macro that combines *xcreat()* and *fdfopen();* its definition is:

```
fdfopen(xcreat(file, mode),1)
```

xfopen(file, mode)
Xfopen() is a macro that combines *xopen()* (see below) and *fdfopen();* its definition is:

```
fdfopen(xopen(file, mode), mode)
```

xlink(f1, f2)
Xlink() is used in the same way as *link*(II). It is an interface to *link*(II) that handles all error conditions. It returns 0 on success, and the value of *fatal()* on failure.

xmsg(file, funcname)
Xmsg() is used by the other *x*-routines to generate an error message based on *errno* (see *INTRO*(II)). It calls *fatal()* with the appropriate error message. The second argument is a pointer to the calling function's name (a string). There are predefined messages for the most common errors. Other errors cause a message of the form:

str = sprintf(space, "error = %d, function = '%s'", errno, funcname)

to be passed to *fatal()*.

xopen(name, mode)

Xopen() is used in the same way as *open*(II). It is an interface to *open*(II) that handles all error conditions. It returns a file descriptor on success, and the value of *fatal()* on failure.

xpipe(t)

Xpipe() is used in the same way as *pipe*(II). It is an interface to *pipe*(II) that handles all error conditions. It returns 0 on success, and the value of *fatal()* on failure.

xunlink(f)

Xunlink() is used in the same way as *unlink*(II). It is an interface to *unlink*(II) that handles all error conditions. It returns 0 on success, and the value of *fatal()* on failure.

2.4 Write Set

write(fildes, buffer, nbytes)

Write() is used in the same way as *write*(II). It is an interface to *syswrite()* (see below) that handles all error conditions. It returns the number of bytes written on success, and the value of *fatal()* on failure.

syswrite(fildes, buffer, nbytes)

Syswrite() is identical to *write*(II), except that the name *write* has been changed to *syswrite*.

Guide to IBM Remote Job Entry for PWB/UNIX Users

A. L. Sabsevitz

Bell Laboratories
Piscataway, New Jersey 08854

1. PREFACE

A set of background processes supports remote job entry (RJE) from a PWB/UNIX* computer to IBM System/360 and /370 host computers. "Hasp" is the common name used for the collection of programs and for the file organization that provides this facility; it allows PWB/UNIX to communicate with IBM's Job Entry Subsystem by mimicking an IBM 2770 remote station. The *PWB/UNIX User's Manual* page *hasp*(VIII) summarizes their design and operating procedures. That manual also contains a terse description of the *send*(I) command, which is the user's primary interface to RJE.[1] These are the definitive sources for information about RJE. Although the word "Hasp" may be used in this guide, it represents all IBM RJE subsystems of the PWB/UNIX system.

This guide is a tutorial overview of RJE.[2] It is addressed to the user who needs to know how to use the system, but does *not* need to know details of its implementation. The two following sections constitute an introduction to RJE.

2. PRELIMINARIES

To become a PWB/UNIX user, you must receive a login name that identifies you to the PWB/UNIX system. You should also get a copy of the *PWB/UNIX User's Manual.* This is a fairly complete description of the system and includes a section entitled "How to Get Started," which introduces you to PWB/UNIX; you should read that section before proceeding with this guide.

In order to begin using RJE, you need only become familiar with a subset of basic commands. You must understand the directory structure of the file system, and you should know something about the attributes of files: see *chdir*(I), *chmod*(I), *chown*(I), *cp*(I), *ln*(I), *ls*(I), *mkdir*(I), *mv*(I), *rm*(I). You must know how to enter, edit, and examine text files: see *cat*(I), *ed*(I), *pr*(I). You should know how to communicate with other users and with the system: see *mail*(I), *mesg*(I), *who*(I), *write*(I). And, finally, you might have to know how to describe your terminal to the system: see *ascii*(V), *stty*(I), *tabs*(I).

3. BASIC RJE

Let's suppose that you have used the editor, *ed*(I), to create a file "jobfile" that contains your control statements (JCL) and input data. This file should look exactly like a card deck, except that alphabetic characters, for convenience, may be in either upper or lower case. Here is an example:

```
% cat jobfile
//gener job (9999,r740),pgmrname,class=x usr=(mylogin,myplace)
//step exec pgm=iebgener
//sysprint dd sysout=a
//sysin dd dummy
//sysut2 dd sysout=a
//sysut1 dd *
   first card of data
       :
       :
   last card of data
/*
```

---

* UNIX is a Trademark of Bell Laboratories.

1. In this paper, RJE refers to the PWB/UNIX facilities provided by *hasp*(VIII), and *not* to the Remote Job Entry feature of IBM's HASP or JES2 subsystems.

2. The original versions of this manual and of RJE itself were written by T. G. Lyons.

To submit this job for execution, you must invoke the *send*(I) command:

```
% send jobfile
```

The system will reply:

```
10 cards.
Queued as /usr/hasp/xmit311.
```

Note that *send* tells you how many cards it submitted and reports the position that your job has been assigned in the queue of all jobs waiting to be transmitted to the host system. Until the transmission of the job actually begins, you can prevent the job from being transmitted by doing a "chmod 0" on the queued file to make it unreadable. For our example, you could say: "chmod 0 /usr/hasp/xmit311".

When your job is accepted by the host system, a job number will be assigned to it, and an acknowledgement message will be generated. This indicates that your job has been scheduled on the host system. Later, after the job has executed, its output will be returned to the PWB/UNIX system. You will be notified automatically of both of these events: if you are logged in when RJE detects these events, and if you are permitting messages to be sent to your terminal (see *mesg*(I)), the following two messages will be sent to you (still using the example above) when the job is scheduled and when the output is returned, respectively:

```
        Two bells
$12.18.42 JOB 384 ON RM4.RD1  --  GENER     PGMRNAME
        Bell

        Two bells
12:21:54 /a1/user/rje/prnt0 384.gener ready
        Bell
```

The job-acknowledgement message is passed on directly from the host system, as indicated by the fact that it appears in upper case. The output-ready message is generated by RJE and appears in lower case. Two bells, with an interval of one second between them, precede each message. They should be interpreted as a warning to stop typing on your terminal, so that the imminent message is not interspersed with your typing.

If you are not logged in when one of these events occurs, or if you do not allow messages to be sent to your terminal, then the notification will be posted to you via the *mail*(I) command. You can prevent messages directly by executing the *mesg*(I) command, or indirectly by executing another command, such as *pr*(I), which prohibits messages for as long as it is active. You may inspect (by invoking the *mail* command) your *.mail* file at any time for messages that have been diverted. For this example, this might look as follows:

```
% mail

From rje Mon Aug  1 12:20:36 1977
$12.18.42 JOB 384 ON RM4.RD1  --  GENER     PGMRNAME

From rje Mon Aug  1 12:21:55 1977
12:21:54 /a1/user/rje/prnt0 384.gener ready

Save?
```

Note that there may exist a discrepancy between the host and PWB/UNIX clocks.

The job-acknowledgement message performs two functions. First, it confirms the fact that your job has been scheduled for eventual execution. Second, it assigns a number to the job in such a way that the number and the name together will uniquely identify the job for some period of time.

The output-ready message provides the name of a PWB/UNIX file into which output has been written and identifies the job to which the output belongs (see *ls*(I)):

```
% ls -l prnt0
-r--r-xr--  1 rje        1184 Aug  1 12:21 prnt0
```

Note that rje retains ownership of the output and allows you only read access to it. It is intended that you will inspect the file, perhaps extract some information from it and then promptly delete it (see *rm*(I)):

```
% rm -f prnt0
```

The retention of machine-generated files, such as RJE output, is discouraged. It is your responsibility to remove files from your RJE directory. Files of RJE output may not exceed 256K bytes. In addition, only files of 64K bytes are guaranteed to be accepted in their entirety. Limits of 64K, 128K, or 192K may be automatically enforced if file space gets scarce. Output beyond the current limit will be discarded, with no provision for retrieving it. The user should also be aware of the fact that RJE attempts to keep roughly 1000 "blocks" free on any file system it uses. Warning messages or suspension of certain functions will occur as this limit is approached.

The most elementary way to examine your output is to *cat* it to your terminal. The Appendix shows the result of listing the output of our sample job in this way. Printouts are stored with standard tabs to conserve space, so you must ensure that the tabs are set on your terminal at every eighth column across the entire line; *tabs*(I) will do that for you.[3] Because PWB/UNIX has no high-volume printing capability, you should route to the host's printer any large listings of which you desire a hard copy.

The structure of an output listing will generally conform to the following sequence:

```
HASP log
jcl information
data sets
HASP end
```

"Burst" pages are discarded. Single, double, and triple spacing is reflected in the output file, but other forms controls, such as the skip to the top of a new page, are suppressed. Page boundaries are indicated by the presence of a space *character* at the end of the last line of each page.

The big file scanner *bfs*(I) or the context editor *ed*(I) provide a more flexible method than *cat*(I) for examining printed output; *bfs* can handle files of any size and is more efficient than *ed* for scanning files.

RJE is also capable of receiving punched output as formatted files (see *ebcdic*(V)); this format allows an exact representation of an arbitrary card deck to be stored on the PWB/UNIX machine. However, there are few commands that can be used to manipulate EBCDIC files. You will probably want to route your punched output to one of the host's output devices.

4. SEND COMMAND

The *send*(I) command is capable of more general processing than has been indicated in the previous section. In the first place, it will concatenate a sequence of files to create a single job stream. This allows files of JCL and files of data to be maintained separately on the PWB/UNIX machine. In addition, it recognizes any line of an input file that begins with the character "~" as being a *control* line that can call for the inclusion, inside the current file, of some other file. This allows you to "send" a top-level skeleton that "pulls" in subordinate files as needed. Some of these may be "virtual" files that actually consist of the output of PWB/UNIX commands or Shell procedures. Furthermore, the *send* command is able to collect input directly from a terminal, and can be instructed to prompt for required information.

Each source of input can contain a format specification that determines such things as how to expand tabs and how long can an input line be. The manual page for *fspec*(V) explains how to define such formats. When properly instructed, *send* will also replace arbitrarily defined "keywords" by other text strings or by EBCDIC character codes. (These two substitution facilities are useful in other applications besides RJE; for that reason, *send* may be invoked under the name *gath* to produce standard output *without* submitting an RJE job.)

---

3. If your terminal doesn't have tabs, you should use the *stty*(I) command to cause PWB/UNIX to automatically convert tabs to spaces on output to the terminal.

Two aspects of *send* with which everyone should be acquainted are the ability to specify to which computer a job is to be submitted, and the ability to verify a job prior to submission. To run our sample job on a host machine known to RJE as "A", we would issue the command:

```
% send A jobfile
```

When no host is explicitly cited, *send* makes a reasonable choice.

To verify the text of a collected job stream, without actually submitting it, set the "-lq" flags:

```
% send -lq jobfile
```

The complete list of arguments and flags that control the execution of *send* can be found in *send*(I).

5. JOB STREAM

It is assumed that the job stream submitted as the result of a single execution of *send* consists of a single *job*, i.e., the file that is queued for transmission should contain one JOB card near the beginning and no others. A priority control card may legitimately precede the JOB card. The JOB card must conform to the local installation's standard. At BISP, it has the following structure:

```
//name job (acct[,...]),pgmrname[,keywds=?] [usr=...]
```

6. USER SPECIFICATION

The "usr=..." field is required if any print or punch output is to be delivered to the PWB/UNIX user.

```
usr=(login,place[,[level][,retry]])
```

where *login* is the PWB/UNIX login name of the user, *level* is the desired level of notification (see end of this section for an explanation), *retry* is a one-character code specifying the number of attempts to retransmit an entire job if the transmission to the host computer is interrupted by an unrecoverable error (default is three attempts; the digits "1" through "9" specify that number of retries; "0", "y", or "Y" invoke the default; any other entry in this field limits the number of attempts to one, i.e., no retry), and *place* is as follows:

A. If *place* is the name of a *directory* (writable by others), then the output file is placed there as a unique *prnt* or *pnch* file (up to 500 of each allowed). The mode of the file will be 454.

B. If *place* is the name of an existing, writable (by others), non-executable (by others) file, then the output file replaces it. The mode of the file will be 454.

C. If *place* is the name of a non-existent file in a writable (by others) directory, then the output file is placed there. The mode of the file will be 454.

D. If *place* is the name of an executable (by others) file, then the RJE output is set up as standard input to *place*, and *place* is executed. Five string arguments are passed to *place*. For example, if *place* is a shell procedure, the following arguments are passed as $1 ... $5:

   1. Flag indicating whether file space is scarce in the file system where *place* resides. 0 indicates that space is *not* scarce, while 1 indicates that it is.
   2. Job name.
   3. Programmer's name.
   4. Job number.
   5. Login name from the "usr=..." specification.

   A ":" is passed if a value is not present.

E. In all other cases, the output will be thrown away.

The *place* value must not be a full pathname, unless it refers to an executable file (see D above). For cases A, B, and C above (and case D, if a full pathname is not supplied), the name of the user's login directory will be used to form a full pathname.

The "usr=..." field may occur anywhere within the first 100 card images sent and within the first 200 output images received by the PWB/UNIX system. The only restrictions are:

- Column one must contain a "/" or a "*".
- "usr=..." must begin after column 4 and must be preceded by a space.

Therefore, the "usr=..." field may be placed on the JOB card, a comment card, passed as data, etc.

For redirection of output by the host, a "usr=..." card, if not already present, must be supplied by the user. This can be done by placing a job step that creates this card before your output steps.

Messages generated by RJE or passed on from the host are assigned a level of importance ranging from 1 to 9. The levels currently in use are:

3 transmittal assurance
5 job acknowledgement
6 output ready message
7 transmit format error

The optional "level" field of the "usr=..." specification must be a one- or two-digit code. A message from the host with importance "x" (where x comes from the above list) is compared with each of the two decimal digits "mw". If $x \geqslant w$ and if the user is logged in and is accepting messages, the message will be written to his or her terminal. Otherwise, if $x \geqslant m$, the message will be mailed to the user. In all other cases, the message will be discarded. The default "level field" is "54". You should specify level "1" if you want to receive complete notification, and level "59" to divert the last three messages in the above list to your mailbox.

7. CONTROL CARDS

A number of control cards are recognized by the host's HASP subsystem. Two are of particular interest to RJE users, because they control the disposition of output. These are the ROUTE and OUTPUT cards, and their use is illustrated below. If used, both should be inserted into a job stream *immediately* after the JOB card.

The ROUTE card can be used to direct the entire printed or punched output of a job to a specified destination. Two cards are required to direct both outputs:

```
/*route      punch     local
/*route      print     rmt55
```

The ROUTE card has a fixed format. "Print" or "punch" must begin in column 10, and the destination field in column 16.

The proper use of the OUTPUT card is a bit more complicated. It allows you to associate parameters with all SYSOUT data sets whose forms numbers match the one specified on the OUTPUT card. The forms number is fictitious and may consist of up to four characters. A copy count and destination are among the parameters that may be associated with SYSOUT data sets in this way:

```
//name job ...
/*output py d=rmt56
/*output abcd d=local,n=2
//step exec ...
//prt1 dd sysout=a
//prt2 dd sysout=(a,,abcd)
//prt3 dd sysout=(a,,py)
//pnch dd sysout=(b,,abcd)
```

In the above example, one copy of *prt1* would be directed to the default destination and one copy of *prt3* to *rmt56*. Two copies each would be made of *prt2* and *pnch*, and they would remain at the *local* site.

8. MONITORING RJE

RJE is designed to be an autonomous facility that does not require manual supervision. RJE is initiated by the PWB/UNIX operator after system "reboots" and continues in execution indefinitely. Experience has proved it to be reasonably robust, although it is vulnerable to system crashes and reconfigurations.

Users have a right to assume that, if the PWB/UNIX system is up for production use, RJE should also be up. This implies more than an ability to execute the *send*(I) command, which should be available at all times. It means that queued jobs should be submitted to the host for execution and their output returned to the PWB/UNIX system. If a user cannot obtain any throughput from RJE, the user should so advise the PWB/UNIX operators.

The *rjestat*(I) command, invoked without the "-" argument, will report the status of all RJE links for which a given PWB/UNIX system is configured. It may sometimes also print a message of the day from RJE.

```
% rjestat

15:12:24 RJE to B is operating normally.

15:12:25 A is not responding to RJE.
 (8 files queued since 14:34:26)
```

A parenthetical statement, such as the last line above, will summarize any backlog of queued files waiting to be transmitted to the host machine. A backlog that persists for 20 minutes or more often is an indication that there exists a problem with the corresponding RJE link.

A host machine may be reported to be not responding to RJE because it is down, or because of its operator's failure to initialize the associated line, or because of a communications hardware failure.

Appendix—Sample Output Listing

```
% cat rje/prnt0
14.40.31 JOB 384  $HASP373 GENER  STARTED - INIT 26 - CLASS X - SYS RRMA
14.40.32 JOB 384  $HASP395 GENER  ENDED


------ JES2 JOB STATISTICS ------

 1 AUG 77 JOB EXECUTION DATE

          54 CARDS READ

          76 SYSOUT PRINT RECORDS

           0 SYSOUT PUNCH RECORDS


      0.01 MINUTES EXECUTION TIME
   1        //GENER JOB (9999,R740),PGMRNAME,CLASS=X                    JOB 384
            ***       USR=(MYLOGIN,MYPLACE)
   2        //IEBGENER EXEC PGM=IEBGENER
   3        //SYSPRINT DD DUMMY
   4        //SYSIN DD DUMMY
   5        //SYSUT2 DD SYSOUT=A
   6        //SYSUT1 DD *
            //
IEF236I ALLOC. FOR GENER IEBGENER
IEF237I DMY ALLOCATED TO SYSPRINT
IEF237I DMY ALLOCATED TO SYSIN
IEF237I JES ALLOCATED TO SYSUT2
IEF237I JES ALLOCATED TO SYSUT1
IEF142I GENER IEBGENER - STEP WAS EXECUTED - COND CODE 0000
IEF285I   JES2.JOB0384.SO0102                    SYSOUT
IEF285I   JES2.JOB0384.SI0101                    SYSIN
IEF373I STEP /IEBGENER/ START 77242.1440
IEF374I STEP /IEBGENER/ STOP 77242.1440 CPU  0MIN 00.13SEC SRB 0MIN 00.01SEC VIRT 36K SYS 188K

******* SERVICE UNITS=0000174          SERVICE RATE=0000268 SERVICE UNITS/SECOND
******* PERFORMANCE GROUP=005
******* EXCP COUNT BY UNIT ADDRESS
IEF375I  JOB /GENER / START 77242.1440
IEF376I  JOB /GENER / STOP  77242.1440 CPU    0MIN 00.13SEC SRB    0MIN 00.01SEC

******* SERVICE UNITS=0000174          SERVICE RATE=0000268 SERVICE UNITS/SECOND
******* APPROXIMATE PROCESSING TIME=           .01 MINUTES
******* EXCPS=000000000
******* PROJECTED CHARGES=           .01
```

first line of data

:

last line of data

```
-OS/VS2 REL 3.7 JES2- END  JOBNAME=GENER   BIN=R740       JOB #=384  PGMRNAME
-OS/VS2 REL 3.7 JES2- END  JOBNAME=GENER   BIN=R740       JOB #=384  PGMRNAME
-OS/VS2 REL 3.7 JES2- END  JOBNAME=GENER   BIN=R740       JOB #=384  PGMRNAME
% rm -f rje/prnt0
```

ADDENDUM #1

UNIVAC RJE

I. PREFACE {1.} *

A set of PWB/UNIX † background processes supports remote job entry (RJE) from a PWB/UNIX computer to UNIVAC 1100-series host computers. "Uvac" is the common name used for the collection of programs and for the file organization that provides this facility.

II. BASIC RJE {3.}

Alphabetic characters, for convenience, may be in either upper or lower case. The master space "@" may be represented as "`". Here is an example:

```
% cat jobfile
`run  echo,acct-no,project-id     .    usr= (mylogin,myplace)
`ed,i   .elt
   first card of data
        :
        :
   last card of data
`fin
```

The system will reply:

```
8 cards.
Queued as /usr/uvac/xmit311.
```

The job acknowledgement messages are:

```
        Two bells
$12.18.42 001.ECHO STARTED     ACCT-NO
        Bell

        Two bells
12:21:54 /a1/user/rje/prnt0 .echo ready
        Bell
```

To route your printout to the host's printer, insert the following control card into the run stream following the RUN card:

```
`sym     print$,,pr
```

"Burst" pages are *not* discarded and the reception of punched output is *not* supported.

III. JOB STREAM {5.}

The RUN card must conform to the local installation's standard. In general, it has the following format:

```
@run    name,acct-no,project-id    [ . usr= ...]
```

IV. USER SPECIFICATION {6.}

The "usr= ..." field may occur anywhere within the first 100 card images sent and within the first 200 output images received by the PWB/UNIX system. The only restrictions are:

- Column 1 must contain a "@" or a "`".
- "usr= ..." must begin after column 4 and must be preceded by a space.

Therefore, the "usr= ..." field may be placed on the RUN card, a message card, passed as data, etc.

---

* Numbers enclosed in curly braces are section numbers of *Guide to IBM Remote Job Entry for PWB/UNIX Users* by A. L. Sabsevitz, September 1977.

† UNIX is a Trademark of Bell Laboratories.

V. CONTROL CARDS [7.]

This section of the guide is not applicable to UNIVAC RJE.

VI. MONITORING RJE [8.]

The interactive status terminal capability of the *rjestat*(I) command is *not* implemented.

SCCS/PWB

User's Manual

L. E. Bonanni
A. L. Glasser

November 1977

Bell Telephone Laboratories, Incorporated

SCCS/PWB
User's Manual

CONTENTS

SCCS/PWB User's Manual

L. E. Bonanni

Bell Laboratories
Piscataway, New Jersey 08854

A. L. Glasser

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

The Source Code Control System (SCCS) is a system for controlling changes to files of text (typically, the source code and documentation of software systems). It provides facilities for storing, updating, and retrieving any version of a file of text, for controlling updating privileges to that file, for identifying the version of a retrieved file, and for recording who made each change, when and where it was made, and why. SCCS is a collection of programs that run under the PWB/UNIX* time-sharing system.

This document, together with the *PWB/UNIX User's Manual* [4], is a complete user's guide to Version 4 of SCCS, and supersedes all previous versions of the SCCS/PWB manual; it covers the following topics:

- How to get started with SCCS.
- The version numbering scheme.
- Basic information needed for day-to-day use of SCCS commands, including a discussion of the more useful arguments.
- Protection and auditing of SCCS files, including the differences between the use of SCCS by *individual* users on one hand, and *groups* of users on the other.

Neither the implementation of SCCS nor the installation procedure for SCCS are described here.

1. INTRODUCTION

The Source Code Control System (SCCS) is a collection of PWB/UNIX [1] commands that help individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to conceive of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, and records who made each change, when and where it was made, and why. This is important in environments in which programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores on disk the original file and, whenever changes are made to it, stores only the *changes;* each set of changes is called a "delta."

This document, together with the *PWB/UNIX User's Manual* [4], is a complete user's guide to Version 4 of SCCS. This manual contains the following sections:

- *SCCS for Beginners:* How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- *How Deltas Are Numbered:* How versions of SCCS files are numbered and named.
- *SCCS Command Conventions:* Conventions and rules generally applicable to all SCCS commands.
- *SCCS Commands:* Explanation of all SCCS commands, with discussions of the more useful arguments.

* UNIX is a Trademark of Bell Laboratories.

- *SCCS Files:* Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

2. SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a PWB/UNIX system, create files, and use the text editor [2,3]. A number of terminal-session fragments are presented below. All of them should be tried: the best way to learn SCCS is to use it.

To supplement the material in this manual, the detailed SCCS command descriptions (appearing in alphabetical order in Section I of [4]) should be consulted. Section 5 below contains a list of all the SCCS commands. For the time being, however, only basic concepts will be discussed.

2.1 Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *SCCS IDentification* string (SID), composed of at most four components, only the first two of which will concern us for now; these are the "release" and "level" numbers, separated by a period. Hence, the first delta is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed (usually, this indicates a major change to the file) as discussed below.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2 Creating an SCCS File—The "admin" Command

Consider, for example, a file called "lang" that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

We wish to give custody of this file to SCCS. The following *admin* command (which is used to *administer* SCCS files) creates an SCCS file and initializes delta 1.1 from the file "lang":

```
admin -ilang s.lang
```

All SCCS files *must* have names that begin with "s.", hence, "s.lang". The **-i** keyletter, together with its value "lang", indicates that *admin* is to create a new SCCS file and *initialize* it with the contents of the file "lang". This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The *admin* command replies:

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described in Section 5.1 below.

The file "lang" should be removed (because it can be easily reconstructed by using the *get* command, below):

```
rm lang
```

2.3 Retrieving a File—The "get" Command

The command:

```
get s.lang
```

causes the creation (retrieval) of the latest version of file "s.lang", and prints the following messages:

```
1.1
5 lines
No id keywords (cm7)
```

This means that *get* retrieved version 1.1 of the file, which is made up of 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file; hence, the file "lang" is created.

The above *get* command simply creates the file "lang" read-only, and keeps no information whatsoever regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the *delta* command (see below), the *get* command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The -e keyletter causes *get* to create a file "lang" for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the *p-file*, that will be read by the *delta* command. The *get* command prints the same messages as before, except that the warning message is not issued.

The file "lang" may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

2.4 Recording Changes—The "delta" Command

In order to record within the SCCS file the changes that have been applied to "lang", execute:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made; for example:

```
comments? added more languages
```

Delta then reads the *p-file*, and determines what changes were made to the file "lang". It does this by doing its own *get* to retrieve the original version, and by applying *diff*(I)[1] to the original version and the edited version.

When this process is complete, at which point the changes to "lang" have been stored in "s.lang", *delta* outputs:

```
No id keywords (cm7)
1.2
2 inserted
0 deleted
5 unchanged
```

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file "s.lang".

---

1. All references of the form *name*(*N*) refer to item *name* in section *N* of the *PWB/UNIX User's Manual* [4].

2.5 More about the "get" Command

As we have seen:

```
get s.lang
```

retrieves the latest version (now 1.2) of the file "s.lang". This is done by starting with the original version of the file and successively applying deltas (the changes) in order, until all have been applied.

For our example, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** keyletter are SIDs (see Section 2.1 above). Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the *highest* level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the *release* number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with the *get* command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, *get* retrieves the latest version *before* release 2; it also interprets this as a request to change the release number of the delta we wish to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*. *Get* then outputs:

```
1.2
7 lines
```

indicating the retrieval of version 1.2. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and *delta* executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

we will see, by *delta's* output, that version 2.1 is indeed created:

```
No id keywords (cm7)
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

2.6 The "help" command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message, and may be used to obtain a fuller explanation of that message by use of the *help* command:

```
help co1
```

This produces the following output:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, *help* is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

3. HOW DELTAS ARE NUMBERED

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree, in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the *release* number when making a delta, to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas, unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 1.

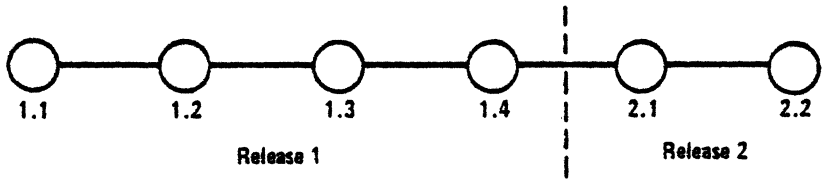


Figure 1. Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. It represents the normal *sequential* development of an SCCS file, in which changes that are part of any given delta are dependent upon *all* the preceding deltas.

However, there are situations in which it is necessary to cause a *branching* in the tree, in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3, and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a "branch" of the tree, and its name consists of *four* components, namely, the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers, as follows:

```
release.level.branch.sequence
```

The *branch* number is assigned to each branch that is a descendant of a particular trunk delta, with the first such branch being 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a *particular branch.* Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 2.

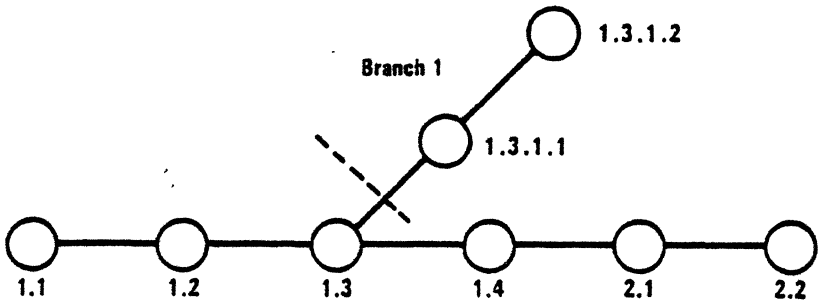


Figure 2. Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the *entire* path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n.* If a delta on this branch then has another branch emanating from *it,* all deltas on the new branch will be named 1.3.2.*n* (see Figure 3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the *chronologically* second delta on the *chronologically* second branch whose *trunk* ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

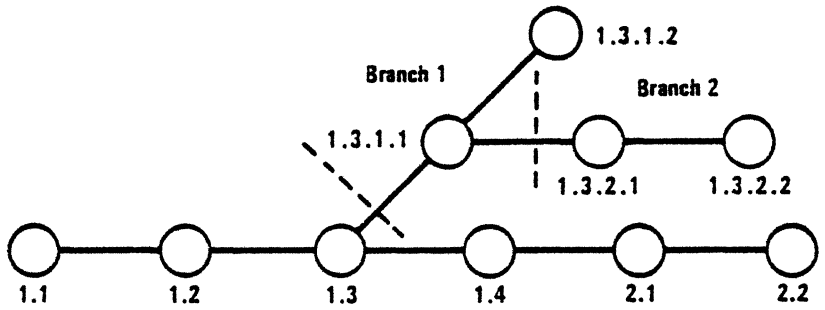


Figure 3. Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

4. SCCS COMMAND CONVENTIONS

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below. SCCS commands accept two types of arguments: *keyletter* arguments and *file* arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (—), followed by a lower-case alphabetic character, and, in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming *all* the SCCS files within the directory. Non-SCCS files and unreadable[2] files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name "—" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the *name* of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines [4] with, for example, the *find*(I) or *ls*(I) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to *all* file arguments of that command. All keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right.

Somewhat different argument conventions apply to the *help, what,* and *sccsdiff* commands (see Sections 5.5, 5.8, and 5.9).

Certain actions of various SCCS commands are controlled by *flags* appearing in SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see *admin*(I).

The distinction between the *real user* and the *effective user* of a PWB/UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a PWB/UNIX system); this subject is further discussed in Section 6.1.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file,* which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, is given the same mode (see *chmod*(I)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file,* called the *z-file,* whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the *process number* [1] of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files;* they may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on the diagnostic output [5]) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The *code* in parentheses may be used as an argument to the *help* command (see Section 5.5) to obtain a further explanation of the diagnostic message.

Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of *that* file and to proceed with the next file, in order, if more than one file has been named.

5. SCCS COMMANDS

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in [4], and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

2. Because of permission modes (see *chmod*(I)).

Because the commands *get* and *delta* are the most frequently used, they are presented first. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.
prt	Formats and prints portions of SCCS files.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
chghist	Changes the commentary associated with a delta.
what	Searches any PWB/UNIX file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the *get* command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.

5.1 get

The *get* command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file;* its name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory [1] and is owned by the real user. The mode assigned to the *g-file* depends on how the *get* command is invoked, as discussed below.

The most common invocation of *get* is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree, and produces (for example) on the standard output [5]:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file (see Section 5.1.1 for a discussion of ID keywords).

The generated *g-file* (file "abc") is given mode 444 (read-only), since this particular way of invoking *get* is intended to produce *g-files* only for inspection, compilation, etc., and *not* for editing (i.e., *not* for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)

5.1.1 ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file,* so as to have this information appear in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification (ID) keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example:

%I%

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%M%** is defined as the name of the *g-file.* Thus, executing *get* on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by *get,* the following message is issued:

No id keywords (cm7)

This message is normally treated as a warning by *get,* although the presence of the **i** flag in the SCCS file causes it to be treated as an error (see Section 5.2 for further information).

For a complete list of the approximately twenty ID keywords provided, see *get*(1).

5.1.2 Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **–r** keyletter of *get.*

The **–r** keyletter is used to specify an SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example:

```
get –r1.3 s.abc
```

retrieves version 1.3 of file "s.abc", and produces (for example) on the standard output:

1.3
64 lines

A branch delta may be retrieved similarly:

```
get –r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

1.5.2.3
234 lines

When a two- or four-component SID is specified as a value for the **–r** keyletter (as above) and the

particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the *trunk* delta with the highest level number within the given release, if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, *get* retrieves the *trunk* delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file "s.abc", and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file "s.abc" below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch, if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The **-t** keyletter is used to retrieve the latest ("top") version in a particular *release* (i.e., when no **-r** keyletter is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

5.1.3 Retrieval with Intent to Make a Delta

Specification of the **-e** keyletter to the *get* command is an indication of the intent to make a delta, and, as such, its use is restricted. The presence of this keyletter causes *get* to:

1. Check the *user list* (which is the list of *login* names of users allowed to make deltas (see Section 6.2)) to determine if the login name of the user executing *get* is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.
2. Check that the *release* (R) of the version being retrieved satisfies the relation:

   floor $\leqslant$ R $\leqslant$ ceiling

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.

A failure of either condition causes the processing of that SCCS file to terminate.

If the above checks succeed, the **-e** keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a *writable g-file* already exists, *get* terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are *not* substituted by *get* when the **-e** keyletter is specified, because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, *get* does not need to check for the presence of ID keywords within the *g-file*, so that the message:

```
No id keywords (cm7)
```

is never output when *get* is invoked with the **-e** keyletter.

In addition, the **-e** keyletter causes the creation (or updating) of a *p-file*, which is used to pass information to the *delta* command (see Section 5.1.4).

The following is an example of the use of the **-e** keyletter:

```
get -e s.abc
```

which produces (for example) on the standard output:

```
1.3
67 lines
```

If the **-r** and/or **-t** keyletters are used together with the **-e** keyletter, the version retrieved for editing is as specified by the **-r** and/or **-t** keyletters.

The keyletters **-i** and **-x** may be used to specify a list (see *get*(I) for the syntax of such a list) of deltas to be *included* and *excluded*, respectively, by *get*. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be *not* applied. This may be used to undo, in the version of the SCCS file to be created, the effects of a previous delta. Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. (Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*.) Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists, and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

☛ *The* **-i** *and* **-x** *keyletters should be used with extreme care.*

The **-k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of *get* with the **-e** keyletter, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **-k** keyletter is identical to one produced by *get* executed with the **-e** keyletter. However, no processing related to the *p-file* takes place.

5.1.4 The p-file and Concurrent Deltas

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of *get* commands with the **-e** keyletter may be executed on the same file, provided that no two executions retrieve the same version *nor* lead to the subsequent creation of the same version by *delta*.

The *p-file* (which is created by the *get* command invoked with the **-e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":[3]

3. Other information may be present, but is not of concern here. See *get*(I) for further discussion.

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get.*

The first execution of "get —e" causes the *creation* of the *p-file* for the corresponding SCCS file. Subsequent executions only *update* the *p-file* by inserting a line containing the above information. Before inserting this line, however, *get* checks that:

- No entry already in the *p-file* specifies as already retrieved the SID of the version to be retrieved.
- That the new ("to-be-created") SID is not already specified as such in the *p-file.*

If both checks succeed, the user is informed that other deltas are in progress, and processing continues. If either check fails, an error message results. It is important to note that the various executions of *get* should be carried out from different directories. Otherwise, only the first execution will succeed, since subsequent executions would attempt to over-write a *writable g-file,* which is an SCCS error condition. In practice, such multiple executions are performed by different users,[4] so that this problem does not arise, since each user normally has a different working directory [5].

Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved by *get,* as well as the SID of the version to be eventually created by *delta,* as a function of the SID specified to *get.*

5.1.5 Keyletters That Affect Output

Specification of the —p keyletter causes *get* to write the retrieved text to the standard output, rather than to a *g-file.* In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get —p s.abc > arbitrary-filename
```

The —p keyletter is particularly useful when used with the "!" or "$" arguments of the PWB/UNIX *send*(I) command. For example:

```
send MOD= s.abc REL= 3 compile
```

if file "compile" contains:

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
~—s
~!get —p —rREL MOD
/*
//
```

will *send* the highest level of release 3 of file "s.abc". Note that the line "~—s", which causes *send*(I) to make ID keyword substitutions before detecting and interpreting control lines, is necessary if *send*(I) is to substitute "s.abc" for MOD and "3" for REL in the line "~!get —p —rREL MOD".

The —s keyletter suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent non-diagnostic messages from appearing on the user's terminal, and is often used in conjunction with the —p keyletter to "pipe" the output of *get,* as in:

```
get —p —s s.abc | nroff
```

The —g keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get —g —r4.3 s.abc
```

4. See Section 6.1 for a discussion of how different users are permitted to use SCCS commands on the same files.

TABLE 1. Determination of New SID

Case	*SID Specified**	*−b Keyletter Used†*	*Other Conditions*	*SID Retrieved*	*SID of Delta to be Created*
1.	none‡	no	R defaults to mR	mR.mL	mR.(mL + 1)
2.	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB + 1).1
3.	R	no	R > mR	mR.mL	R.1§
4.	R	no	R = mR	mR.mL	mR.(mL + 1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
7.	R	—	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB + 1).1
8.	R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB + 1).1
9.	R.L	no	No trunk successor	R.L	R.(L + 1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB + 1).1
11.	R.L	—	Trunk successor in release ⩾ R	R.L	R.L.(mB + 1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS + 1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB + 1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S + 1)
15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB + 1).1
16.	R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB + 1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB + 1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

† The **−b** keyletter is effective only if the **b** flag (see *admin*(I)) is present in the file. In this table, an entry of "—" means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the *first* delta in a *new* release.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

This outputs the given SID if it exists in the SCCS file, or it generates an error message, if it does not. Another use of the **−g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get −e −g s.abc
```

The **−l** keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (whose format is described in *get*(I)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get −r2.3 −l s.abc
```

generates an *l-file* showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of "p" with the **−l** keyletter, as in:

```
get −lp −r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. Note that the **−g** keyletter may be used with the **−l** keyletter to suppress the actual retrieval of the text.

The **-m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **-n** keyletter causes each line of the generated *g-file* to be preceded by the value of the %M% ID keyword (see Section 5.1.1) and a tab character. The **-n** keyletter is most often used in a pipeline with *grep*(I). For example, to find, in the latest version of each SCCS file in a directory, all lines that match a given pattern, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both the **-m** and **-n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of the **-n** keyletter), followed by the line in the format produced by the **-m** keyletter. Because use of the **-m** keyletter and/or the **-n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta. Therefore, neither the **-m** keyletter nor the **-n** keyletter may be specified together with the **-e** keyletter.

See *get*(I) for a full description of additional *get* keyletters.

5.2 delta

The *delta* command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and, therefore, a new version of the file.

Invocation of the *delta* command requires the existence of a *p-file* (see Sections 5.1.3 and 5.1.4). *Delta* examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. *Delta* also performs the same permission checks that *get* performs when invoked with the **-e** keyletter. If all checks are successful, *delta* determines what has been changed in the *g-file*, by comparing it (via *diff*(I)) with its own, temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal *get* at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing *delta*, because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed *get* with the **-e** keyletter more than once on the same SCCS file), the **-r** keyletter must be used with *delta* to specify the SID that is to be used by the internal *get* to obtain the *d-file*. The SID specified must, of course, appear in one of the entries in the *p-file;* this entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of *delta* is:

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a newline character. The user's response may be up to 512 characters long, with newlines *not* intended to terminate the response escaped by "\".

If the SCCS file has a **v** flag, *delta* first prompts with:

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR[5] numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?".

5. In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

The **-y** and/or **-m** keyletters are used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input. For example:

```
delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **-m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when *delta* is executed from within a *Shell procedure* (see *sh*(I)).

The commentary (comments and/or MR numbers), whether solicited by *delta* or supplied via keyletters, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of *delta*. This implies that if *delta* is invoked with more than one file argument, and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, *delta* outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by *delta* do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and *delta* is likely to find a description that differs from the user's perception. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If, in the process of making a delta, *delta* finds no ID keywords in the edited *g-file*, the message:

```
No id keywords (cm7)
```

is issued after the prompts for commentary, but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values, or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a *get* without the **-e** keyletter (recall that ID keywords are replaced by *get* in that case), or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file, indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*.[6] If there is only *one* entry in the *p-file*, then the *p-file* itself is removed.

In addition, *delta* removes the edited *g-file*, unless the **-n** keyletter is specified. Thus:

```
delta -n s.abc
```

will keep the *g-file* upon completion of processing.

The **-s** ("silent") keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes *delta* neither to read the standard input nor to write the standard output.

---

6. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file*, which is described in Section 4 above.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by *diff*(1).

5.3 admin

The *admin* command is used to *adminster* SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files, and are discussed in Section 6.3 below.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the *admin* command upon that file.

5.3.1 Creation of SCCS Files

An SCCS file may be created by executing the command:

```
admin -ifirst s.abc
```

in which the value ("first") of the **-i** keyletter specifies the name of a file from which the text of the *initial* delta of the SCCS file "s.abc" is to be taken. Omission of the value of the **-i** keyletter indicates that *admin* is to read the standard input for the text of the initial delta. Thus, the command:

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by *admin* as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the **-i** keyletter), the message is treated as an error and the SCCS file is not created. Only *one* SCCS file may be created at a time using the **-i** keyletter.

When an SCCS file is created, the *release* number assigned to its first delta is normally "1", and its *level* number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **-r** keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **-i** keyletter.

5.3.2 Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for *descriptive text* (see Section 6.2) may be initialized or changed through the use of the **-t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file, although its contents may be arbitrary, and it may be arbitrarily long.

When an SCCS file is being created and the **-t** keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file "desc".

When processing an *existing* SCCS file, the **-t** keyletter specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of "desc"; omission of the file name after the **-t** keyletter as in:

```
admin -t s.abc
```

causes the *removal* of the descriptive text from the SCCS file.

The *flags* (see Section 6.2) of an SCCS file may be initialized, changed, or deleted through the use of the **-f** and **-d** keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See *admin*(I) for a description of all the flags. For example, the **v** flag specifies that *delta* is to prompt for Modification Request (MR) numbers, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The **-f** keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin -ifirst -fv -fmmodname s.abc
```

sets the **v** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the *get* command will use to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the **%M%** ID keyword.) Note that several **-f** keyletters may be supplied on a single invocation of *admin*, and that **-f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **-d** keyletter is used to delete a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **-d** keyletters may be supplied on a single invocation of *admin*, and may be intermixed with **-f** keyletters.

SCCS files contain a list (*user list*) of login names of users who are allowed to create deltas (see Sections 5.1.3 and 6.2). This list is empty by default, which implies that *anyone* may create deltas. To add login names to the list, the **-a** keyletter is used. For example:

```
admin -axyz -awql s.abc
```

adds the login names "xyz" and "wql" to the list. The **-a** keyletter may be used whether *admin* is creating a new SCCS file or processing an existing one, and may appear several times. The **-e** keyletter is used in an analogous manner if one wishes to remove ("erase") login names from the list.

5.4 prt

Prt is used to format and print on the standard output all or parts of an SCCS file (see Section 6.2), preceded by the file's name. The portions of the file to be printed are selected by specifying certain keyletters, which, together with the output formats they generate, are fully described in *prt*(I). This section only describes briefly the **-d**, **-u**, **-f**, and **-t** keyletters, which are sufficient to print all of the more interesting portions of an SCCS file.

The **-d** keyletter is used to print the *delta table* of an SCCS file. The delta table is that portion of the file that contains information relevant to the creation of each delta of the file, namely the SID of the delta, the date and time of creation, the *login* name of the creator, and the numbers of lines inserted, deleted, and unchanged by the delta. The commentary that is entered when a delta is created is also part of the delta table. Thus, executing the command:

```
prt -d s.abc
```

provides a history of the evolution of the SCCS file. In the absence of *any* keyletters, the **-d** keyletter is assumed.

The **-u** keyletter is used to print the *user list*. The **-f** keyletter causes the printing of all the *flags* of the SCCS file. The **-t** keyletter is used to print the *descriptive text* of the SCCS file (see Section 6.2); this could be used, for example, to generate a complete set of file summaries, by executing:

```
prt -t sccs
```

in which "sccs" is the name of a directory containing the SCCS files.

Although *prt* makes the examination of SCCS files convenient, other PWB/UNIX commands (e.g., *ed*(I), *grep*(I)) can be used to create customized print commands in the form of Shell procedures.

5.5 help

The *help* command prints explanations of SCCS commands and of messages that these commands may print. Arguments to *help*, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, *help* prompts for one. *Help* has no concept of *keyletter* arguments or *file* arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will *not* terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces:

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
   rmdel -rSID name ...
```

5.6 rmdel

The *rmdel* command is provided to allow *removal* of a delta from an SCCS file, though its use should be reserved for those cases in which incorrect, global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, and so on.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed, or be the owner of the SCCS file and its directory.

The **-r** keyletter, which is mandatory, is used to specify the *complete* SID of the delta to be removed (i.e., it must have two components for a trunk delta, and four components for a branch delta). Thus:

```
rmdel -r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, *rmdel* checks that the *release* number (R) of the given SID satisfies the relation:

$$\text{floor} \leqslant R \leqslant \text{ceiling}$$

In addition, the login name of the user must appear in the file's *user list*, or the *user list* must be empty. If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file (see Section 6.2) is changed from "D" (for "delta") to "R" (for "removed").

5.7 chghist

The *chghist* command is used to *change* a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the *rmdel* command, except that the delta to be processed is *not* required to be a leaf delta. For example:

```
chghist -r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The *new* commentary is solicited by *chghist* in the manner of the *delta* command. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing *chghist* and the time of its execution.

5.8 what

The *what* command is used to find identifying information within *any* PWB/UNIX file whose name is given as an argument to *what*. Directory names and a name of "—" (a lone minus sign) are *not* treated specially, as they are by other SCCS commands, and no *keyletters* are accepted by the command.

What searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the **%Z%** ID keyword (see *get*(I)), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), newline, or (non-printing) NUL character. Thus, for example, if the SCCS file "s.prog.c" (which is a C program), contains the following line (the %M% and %I% ID keywords were defined in Section 5.1.1):

```
char id[] "%Z%%M%:%I%";
```

and then the command:

```
get —r3.4 s.prog.c
```

is executed, and finally the resulting *g-file* is compiled to produce "prog.o" and "a.out", then the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
   prog.c:3.4
prog.o:
   prog.c:3.4
a.out:
   prog.c:3.4
```

The string searched for by *what* need not be inserted via an ID keyword of *get;* it may be inserted in any convenient manner.

5.9 sccsdiff

The *sccsdiff* command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **—r** keyletter, whose format is the same as for the *get* command. The two versions *must* be specified as the first two arguments to this command in the order in which they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the *pr*(I) command (which actually prints the differences) and must appear before any file names. SCCS files to be processed are named last. Directory names and a name of "—" (a lone minus sign) are *not* acceptable to *sccsdiff*.

The differences are printed in the form generated by *diff*(I). The following is an example of the invocation of *sccsdiff*:

```
sccsdiff —r3.4 —r5.6 s.abc
```

5.10 comb

Comb generates a *Shell procedure* (see *sh*(I)) which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated Shell procedure is written on the standard output.

Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The intended use is for those SCCS files that contain deltas that are so old that they are no longer useful. It is *not* recommended that *comb* be used as a matter of routine; its use should be restricted to a *very* small number of times in the life of an SCCS file.

In the absence of any keyletters, *comb* preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate "middle" deltas on the trunk and on all branches of the tree. Thus, in Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **-p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **-c** keyletter specifies a *list* (see *get*(I) for the syntax of such a list) of deltas to be preserved. All other deltas are discarded.

The **-s** keyletter causes the generation of a Shell procedure, which, when run, produces *only* a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that *comb* be run with this keyletter (in addition to any others desired) *before* any actual reconstructions.

It should be noted that the Shell procedure generated by *comb* is *not* guaranteed to save any space. In fact, it is possible for the reconstructed file to be *larger* than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

6. SCCS FILES

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

6.1 Protection

SCCS relies on the capabilities of the PWB/UNIX operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the *release floor* and *ceiling* flags, and the *user list* (see Section 5.1.3).

New SCCS files created by the *admin* command are given mode 444 (read only). It is recommended that this mode *not* be changed, as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755, which allows only the *owner* of the directory to modify its contents.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files (see Section 6.3). The contents of directories should correspond to convenient logical groupings, e.g., sub-systems of a large project.

SCCS files must have only *one* link (name). The reason for this is that those commands that modify SCCS files do so by creating a temporary copy of the file (called the *x-file*, see Section 4) and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, removing it and renaming the *x-file* would break the link. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user (or a group of users who share the same PWB/UNIX user identification number—user ID—see *passwd*(I)) uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. In addition, when several users share the same user ID (even though they may have different *login* names), all such users have identical file permissions. Therefore, SCCS may be used directly by any one of these users, without any preliminary preparation.

However, there are situations (for example, in large software development projects) in which it is not practical to give the same user ID to all users of SCCS. In these cases, one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the *admin* command). This user is termed the *SCCS administrator* for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel* and *chghist* commands.

The interface program must be owned by the SCCS administrator, and must have the *set user ID on execution* bit on (see *chmod*(I)), so that the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the interface program for the duration of that command's execution. In this manner, the owner of an SCCS file can modify it at will. Other users whose *login* names are in the *user list* for that file (but who are *not* its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *chghist*. The project-dependent interface program, as its name implies, must be custom-built for each project.

6.2 Format

SCCS files are composed of lines of ASCII text[7] arranged in six parts, as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as its type, its SID, date and time of creation, and commentary.
User Names	List of login names of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in *sccsfile*(V); the *checksum* is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files, they may be processed by various PWB/UNIX commands, such as *ed*(I), *grep*(I), and *cat*(I). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when it is desired to simply "look" at the file.

☛ *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

6.3 Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file, or portions of it (i.e., one or more "blocks") can be destroyed. SCCS commands (like most PWB/UNIX commands) issue an error message when a file does not exist. In addition, SCCS commands use the *checksum* stored in the SCCS file to determine whether a file has been *corrupted* since it was last accessed (possibly by having lost one or more blocks, or by having been modified with, for example, *ed*(I)). *No* SCCS command will process a corrupted SCCS file except the *admin* command with the **-h** or **-z** keyletters, as described below.

It is recommended that SCCS files be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the *admin* command with the **-h** keyletter on all SCCS files:

```
admin -h s.file1 s.file2 ...
   or
admin -h directory1 directory2 ...
```

7. Versions of SCCS up to and including Version 3 used non-ASCII files. Therefore, files created by earlier versions of SCCS are incompatible with Version 4 of SCCS.

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

corrupted file (co6)

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect *missing* files. A simple way to detect whether *any* files are missing from a directory is to periodically execute the *ls*(I) command on that directory, and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local PWB/UNIX operations group to request a restoral of the file from a backup copy. In the case of minor damage, repair through use of the editor *ed*(I) may be possible. In the latter case, after such repair, the following command must be executed:

admin −z s.file

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

REFERENCES

[1] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* 17(7):365-75, July 1974.

[2] Kernighan, B. W. UNIX for Beginners. Bell Laboratories, 1973.

[3] Kernighan, B. W. A Tutorial Introduction to the UNIX Text Editor. Bell Laboratories, 1973.

[4] Dolotta, T. A., Haight, R. C., and Piskorik, E. M., eds. *PWB/UNIX User's Manual—Edition 1.0.* Bell Laboratories, May 1977.

[5] Kernighan, B. W., and Ritchie, D. M. UNIX Programming. Bell Laboratories, 1973.

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System[1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, 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. NROFF and TROFF offer unusual freedom in document styling, including: 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.

NROFF and TROFF 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.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is

nroff *options files* (or **troff** *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 file name corresponding to the standard input. If no file names 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:

Option	*Effect*
−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*.
−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 stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
−m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files*.
−r*aN*	Register *a* (one-character) is set to *N*.
−i	Read standard input after the input files are exhausted.
−q	Invoke the simultaneous input-output mode of the **rd** request.

NROFF Only

−T*name* Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 teletype, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

−e Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

−t Direct output to the standard output instead of the phototypesetter.

−f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

−w Wait until phototypesetter is available, if currently busy.

−b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

−a Send a printable (ASCII) approximation of the results to the standard output.

−p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

−g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

nroff **−o***4,8−10* **−T***300S* **−m***abc file1 file2*

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN[2] (for NROFF and TROFF respectively), and the table-construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] 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. TK[4] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

tbl *files* | **eqn** | **troff** **−t** *options* | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT[4] can be used to send TROFF (**−g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

[1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual*, Sixth Edition (May 1975).

[2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.

[3] M. E. Lesk, *Tbl — A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] Internal on-line documentation, on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
1. General Explanation				
2. Font and Character Size Control				
.ps ±*N*	10 point	previous	E	Point size; also \s±*N*.†
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.cs *FNM*	off	-	P	Constant character space (width) mode (font *F*).†
.bd *F N*	off	-	P	Embolden font *F* by *N*−1 units.†
.bd S *F N*	off	-	P	Embolden Special Font when current font is *F*.†
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1-4. Also \f*x*, \f(*xx*, \f*N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position $1 \leqslant N \leqslant 4$.
3. Page Control				
.pl ±*N*	11 in	11 in	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*	0; 26/27 in	previous	v	Page offset.
.ne *N*	-	*N*=1 *V*	D,v	Need *N* vertical space (*V* = vertical spacing).
.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*.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
5. Vertical Spacing				
.vs *N*	1/6in;12pts	previous	E,p	Vertical base line 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*	6.5 in	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*.

---

*Values separated by ";" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of "'" as control character (instead of ".") suppresses the break function.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 ...*	0.8; 0.5in	none	E,m	Tab settings; *left* type, 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	-	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; like **ul** in TROFF.
.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 nobreak 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, 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 *word1 ...*		ignored	-	Exception words.
14. Three Part Titles.				
.tl *'left'center'right'*		-	-	Three part title.
.pc *c*	%	off	-	Page number character.
.lt *±N*	6.5 in	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 anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line use \{*anything*\}.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 $N \leqslant 0$, 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 above forms (like **if**).
.el *anything*		-	-	Else portion of if-else.
17. Environment Switching.				
.ev *N*	*N*=0	previous	-	Environment switched (*push down*).
18. Insertions from the Standard Input				
.rd *prompt*	-	*prompt*=BEL	-	Read insertion.
.ex	-	-	-	Exit from NROFF/TROFF.
19. Input/Output File Switching				
.so *filename*		-	-	Switch source file *(push down).*
.nx *filename*		end-of-file	-	Next file.
.pi *program*		-	-	Pipe output to *program* (NROFF only).
20. Miscellaneous				
.mc *c N*	-	off	E,m	Set margin character *c* and separation *N.*
.tm *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.ig *yy*	-	*.yy*=..	-	Ignore till call of *yy.*
.pm *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.fl	-	-	B	Flush output buffer.
21. Output and Error Messages				

Notes-

- B Request normally causes a break.
- 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.

v,p,m,u Default scale indicator; if not specified, scale indicators are *ignored.*

Alphabetical Request and Section Number Cross Reference

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ (to prevent or delay 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
10.7	\"	Beginning of comment
7.3	\$*N*	Interpolate argument $1 \leqslant N \leqslant 9$
13	\%	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	**x*, *(*xx*	Interpolate string *x* or *xx*
9.1	\a	Non-interpreted leader character
12.3	\b'*abc...*'	Bracket building function
4.2	\c	Interrupt text processing
11.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	\f*x*,\f(*xx*,\f*N*	Change to font named *x* or *xx*, or position *N*
11.1	\h'*N*'	Local horizontal motion; move right *N (negative left)*
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*	Interpolate number register *x* or *xx*
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
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*'	Interpolate width of *string*
5.2	\x'*N*'	Extra line-space function *(negative before, positive after)*
12.2	\z*c*	Print *c* with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(newline)	Concealed (ignored) newline
-	*X*	*X*, any character *not* listed above

The escape sequences \\, \., \", \$, *, \a, \n, \t, and \(newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

Section Reference	*Register Name*	*Description*
3	**%**	Current page number.
11.2	**ct**	Character type (set by *width* 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).
11.3	**hp**	Current horizontal place on *input* line.
15	**ln**	Output line number.
-	**mo**	Current month (1-12).
4.1	**nl**	Vertical position of last printed text base-line.
11.2	**sb**	Depth of string below base line (generated by *width* function).
11.2	**st**	Height of string above base line (generated by *width* function).
-	**yr**	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	**.$**	Number of arguments available at the current macro level.
-	**.A**	Set to 1 in TROFF, if **−a** option used; always 1 in NROFF.
11.1	**.H**	Available horizontal resolution in basic units.
-	**.T**	Set to 1 in NROFF, if **−T** option used; always 0 in TROFF.
11.1	**.V**	Available vertical resolution in basic units.
5.2	**.a**	Post-line extra line-space most recently utilized using **\x'*N*'**.
-	**.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 as physical quadrant (1-4).
4	**.h**	Text base-line high-water mark on current page or diversion.
6	**.i**	Current indent.
6	**.l**	Current line length.
4	**.n**	Length of text portion on previous output line.
3	**.o**	Current page offset.
3	**.p**	Current page length.
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.
11.2	**.w**	Width of previous character.
-	**.x**	Reserved version-dependent register.
-	**.y**	Reserved version-dependent register.
7.4	**.z**	Name 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 ' (acute accent)—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 esthetic reasons. Names must 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 left-parenthesis-introduced, two-character name as in **\n(***xx*.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. 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 Graphic Systems typesetter. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the −T option (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, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6×*S*	*C*
n	En = Em/2	3×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, 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 ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **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.

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 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 + 30 points.

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 affected 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 consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(*xx* where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
´	acute accent	'	close quote
`	grave accent	`	open quote
−	minus	-	hyphen

The characters ´, `, and − may be input by \´, \`, and \− respectively or by their names (Table II). The ASCII characters @, #, ", ´, `, <, >, \, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, 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. The

characters ', `, and _ print as themselves.

2.2. Fonts. The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either \f*x*, \f(*xx*, or \f*N* where *x* and *xx* are the name of a mounted 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. A request for a named but not-mounted font is *ignored*. 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, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 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 imbedding a \s*N* at the desired point to set the size to *N*, or a \s±*N* ($1 \leqslant N \leqslant 9$) 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. The current size is available in the **.s** register. NROFF ignores type size control.

Request Form	*Initial Value*	*If No Argument*	*Notes**	*Explanation*
.ps ±*N*	10 point	previous	E	Point size set to ±*N*. Alternatively imbed \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 is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs *FNM*	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 still or again 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 in the vicinity of 10 points. If *N* is missing the embolden mode is turned off. The column heads above were printed with **.bd** I 3. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.

*Notes are explained at the end of the Summary and Index above.

.bd S *F N*	off	-	P	The characters in the Special Font will be emboldened whenever the current font is *F*. This manual was printed with **.bd S B 3**. The mode must be still or again in effect when the characters are physically printed.
.ft *F*	Roman	previous	E	Font changed to *F*. Alternatively, imbed **\f***F*. The font name **P** is reserved to mean the previous font.
.fp *N F*	R,I,B,S	ignored	-	Font position. This is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

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 $-N$ (N 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 useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register.
.bp $\pm N$	N=1	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request **ns**.
.pn $\pm N$	N=1	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the **%** register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current *left margin* is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches. See §6. 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

*The use of " ' " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

				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* (w.r.t. 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**. Note that 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 |\nR**u.

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 doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a 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 base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with ., ?, or ! is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. 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 imbedded 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 to not look like a control line by prefacing 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 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

.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 (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. Base-line spacing. The vertical spacing *(V)* between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must 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**.

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 imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. 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.

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**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.vs *N*	1/6in;12pts	previous	E,**p**	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x'***N***'** (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set to ±*N*. *N*–1 *Vs (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, the no-space mode inhibits **sp** requests and **bp** requests *without* a next page number. The 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**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ll ±*N*	6.5 in	previous	E,m	Line length is set to ±*N*. In TROFF the maximum (line-length) + (page-offset) is about 7.54 inches.
.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. 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. Strings are created by **ds** and appended to by **as**. 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*. 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*. The 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, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

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 ($1 \leqslant N \leqslant 9$). If an invoked argument doesn't 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 the **\$** was concealed in the definition with a prepended \. The number of currently available arguments is in the **.$** register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and 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 imbed 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 base-line (**.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 §T5). 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 (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.
.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 *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 with respect to 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 found trap 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 don't count). The text may be in-line 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 the same as if the contents of *xx* had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named 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**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 (§11.2) are always arabic.

.rr *R* - ignored - Remove register *R*. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ta *N* ...	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 specifying motion.
.lc *c*	.	none	E	The leader repetition character becomes *c*, or is removed 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 graphic 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*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. \ should not be confused with the ASCII control character ESC of the same name. 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. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	\	\	-	Set escape character to \, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set — **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input (even in NROFF) by \(fi, \(fl, \(ff, \(Fi, and \(Fl respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	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 (normally Times Italic, see §2.2). 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	Underline in NROFF (italicize in TROFF) 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	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 (initially Times Roman).

10.4. Control characters. Both the control character . and the *no-break* control character ' may be changed, if desired. 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	.	.	E	The basic control character is set to *c*, or reset to ".".
.c2 *c*	'	'	E	The *nobreak* 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).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 imbed control lines in a macro created by a diversion.

10.7. 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 many 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 imbedded 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**; it should be noted in this example 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 safely imbedded in *string*, and will not affect the current environment. For example, **.ti -\w'1. 'u** could be used to temporarily indent leftward a distance equal to the size of the string "**1.** ".

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: 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 escape sequence **\k***x* will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction **\kx***word***\h'|\nxu+2u'***word* will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, 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* overprinted with centers aligned; the total width is that of the widest character. *string* should *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 the smallest possible constructed box □.

12.3. Large Brackets. The Special Mathematical Font 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***'** will draw a string of repeated *c*'s towards the right for a distance *N*. (\l is \(lower case L). If *c* looks like a continuation of an expression for *N*, it may 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. In the case of characters that are designed to be connected such as baseline-rule _, underrule ‗, and rooten √, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. 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\1'|0\(rn'\1'|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*' will draw 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 base line. 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 *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1      \"compensate for next automatic base-line 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 done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (\(em), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 *word1* ...		ignored	-	Specify hyphenation points in words with imbedded 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 having the format assigned to register **%**. Any character may be used 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 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 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 **.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).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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.

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 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 $N \leqslant 0$, 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 above forms (like **if**).
.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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ev *N*	*N*=0	previous	-	Environment switched to environment $0 \leqslant N \leqslant 2$. Switching is done in push-down fashion so that restoring a previous environment *must* be done with .ev rather than specific reference.

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* newlines in a row 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*.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 user's terminal. **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 using **nx** (§19); the process would ultimately be ended by an **ex** in the insertion file.

19. Input/Output File Switching

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.so *filename*		-	-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro is not felt until the input level returns to the file level. 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*.
.pi *program*		-	-	Pipe output to *program* (NROFF only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

20. Miscellaneous

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.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 user's terminal.
.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.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; 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. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output 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. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. 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.

The examples to be discussed are intended to be useful and somewhat realistic, but won't 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.

T2. 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 t .tl '\(rn'\(rn' \"troff cut mark
.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 base-line 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 base-line 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. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for 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

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

much as the base-line spacing. The *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
```

T3. 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, base-line 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.

T4. 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.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding 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 diversion 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 diversion
.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 first, include 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 \"it 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 position
.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 base-line 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.

T6. 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 effect another last page.

Table I

Font Style Examples

The following fonts are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The Special Mathematical Font was specially prepared for Bell Laboratories by Graphic Systems, Inc. of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard font available from that company.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |*
● □ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
● ■ — - _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Special Mathematical Font

" ´ \ ^ _ ` ‾ / < > { } # @ + − = *
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ φ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω
√ ‾ ≥ ≤ ≡ ∼ ≃ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ 🕭 | ○ ⎧ ⎩ ⎫ ⎭ ⎨ ⎬ ⎪ ⌊ ⌋ ⌈ ⌉ |

Table II

Input Naming Conventions for ´, `,and ¯ and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

Char	*Input Name*	*Character Name*
’	´	close quote
‘	`	open quote
—	\(em	3/4 Em dash
-	¯	hyphen or
-	\(hy	hyphen
−	\−	current font minus
•	\(bu	bullet
□	\(sq	square
_	\(ru	rule
¼	\(14	1/4
½	\(12	1/2
¾	\(34	3/4

Char	*Input Name*	*Character Name*
fi	\(fi	fi
fl	\(fl	fl
ff	\(ff	ff
ffi	\(Fi	ffi
ffl	\(Fl	ffl
°	\(de	degree
†	\(dg	dagger
’	\(fm	foot mark
¢	\(ct	cent sign
®	\(rg	registered
©	\(co	copyright

Non-ASCII characters and ´, `, _, +, −, =, and • on the special font.

The ASCII characters @, #, ", ´, `, <, >, \, {, }, ~, ^, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	*Input Name*	*Character Name*
+	\(pl	math plus
−	\(mi	math minus
=	\(eq	math equals
*	\(**	math star
§	\(sc	section
´	\(aa	acute accent
`	\(ga	grave accent
_	\(ul	underrule
/	\(sl	slash (matching backslash)
α	\(*a	alpha
β	\(*b	beta
γ	\(*g	gamma
δ	\(*d	delta
ε	\(*e	epsilon
ζ	\(*z	zeta
η	\(*y	eta
θ	\(*h	theta
ι	\(*i	iota

Char	*Input Name*	*Character Name*
κ	\(*k	kappa
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
o	\(*o	omicron
π	\(*p	pi
ρ	\(*r	rho
σ	\(*s	sigma
ς	\(ts	terminal sigma
τ	\(*t	tau
υ	\(*u	upsilon
φ	\(*f	phi
χ	\(*x	chi
ψ	\(*q	psi
ω	\(*w	omega
A	\(*A	Alpha†
B	\(*B	Beta†

Char	Input Name	Character Name
Γ	\(*G	Gamma
Δ	\(*D	Delta
E	\(*E	Epsilon†
Z	\(*Z	Zeta†
H	\(*Y	Eta†
Θ	\(*H	Theta
I	\(*I	Iota†
K	\(*K	Kappa†
Λ	\(*L	Lambda
M	\(*M	Mu†
N	\(*N	Nu†
Ξ	\(*C	Xi
O	\(*O	Omicron†
Π	\(*P	Pi
P	\(*R	Rho†
Σ	\(*S	Sigma
T	\(*T	Tau†
Y	\(*U	Upsilon
Φ	\(*F	Phi
X	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
_	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≅	\(~=	approx =
~	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of

Char	Input Name	Character Name
|	\(br	box vertical rule
‡	\(dd	double dagger
☛	\(rh	right hand
☚	\(lh	left hand
Ⓑ	\(bs	Bell System logo
|	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
▯	\(bv	bold vertical
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

Addendum to the
NROFF/TROFF User's Manual
May 1977

Options

−h (NROFF only) Use output tabs during horizontal spacing to speed up output as well as to reduce output byte count. Device tab settings are assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

−z Efficiently suppresses formatted output. Only message output will occur (from **tm** requests and diagnostics).

Old Requests

.ad c The adjustment type indicator *c* may now also be a number obtained from the ".j" register (see below).

.so name The contents of file *name* will be interpolated at the point the **so** request is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text Prints *text* on the message output and terminates without further processing. If *text* is missing, "User Abort." is printed. Does *not* cause a break. The output buffer is flushed.

New Predefined Number Registers

.k Read-only. Contains the horizontal size of the text portion (without indent) of the current partially-collected output line, if any, in the current environment.

.j Read-only. Indicates the current adjustment mode and type. Can be saved and later given to the **ad** request to restore a previous mode.

.P Read-only. Contains the value 1 if the current page is being printed, and is zero otherwise, i.e., if the current page did *not* appear in the **−o** option list.

.L Read-only. Contains the current line-spacing parameter (the value of the most recent **ls** request).

c. Provides general register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

PWB/MM
Programmer's Workbench
Memorandum Macros

D. W. Smith
J. R. Mashey

October 1977

Bell Telephone Laboratories, Incorporated

PWB/MM
Programmer's Workbench Memorandum Macros

CONTENTS

PWB/MM—Programmer's Workbench Memorandum Macros

D. W. Smith

Bell Laboratories
Piscataway, New Jersey 08854

J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1 Purpose

This memorandum is the user's guide and reference manual for PWB/MM (or just **-mm**), a general-purpose package of text formatting macros for use with the UNIX* text formatters *nroff* [9] and *troff* [9]. The purpose of PWB/MM is to provide to the users of PWB/UNIX a unified, consistent, and flexible tool for producing many common types of documents. Although PWB/UNIX provides other macro packages for various *specialized* formats, PWB/MM has become the standard, general-purpose macro package for most documents.

PWB/MM can be used to produce:

- Letters.
- Reports.
- Technical Memoranda.
- Released Papers.
- Manuals.
- Books.
- etc.

The uses of PWB/MM range from single-page letters to documents of several hundred pages in length, such as user guides, design proposals, etc.

1.2 Conventions

Each section of this memorandum explains a single facility of PWB/MM. In general, the earlier a section occurs, the more necessary it is for most users. Some of the later sections can be completely ignored if PWB/MM defaults are acceptable. Likewise, each section progresses from normal-case to special-case facilities. We recommend reading a section in detail only until there is enough information to obtain the desired format, then skimming the rest of it, because some details may be of use to just a few people.

Numbers enclosed in curly brackets ({}) refer to section numbers within this document. For example, this is {1.2}.

Sections that require knowledge of the formatters {1.4} have a bullet (•) at the end of the section heading.

In the synopses of macro calls, square brackets ([]) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

A reference of the form *name*(N) points to page *name* in section *N* of the *PWB/UNIX User's Manual* [1].

The examples of *output* in this manual are as produced by *troff*; *nroff* output would, of course, look somewhat different (Appendix D shows *both* the *nroff* and *troff* output for a simple letter). In those

* UNIX is a Trademark of Bell Laboratories.

cases in which the behavior of the two formatters is truly different, the *nroff* action is described first, with the *troff* action following in parentheses. For example:

The title is underlined (bold).

means that the title is underlined in *nroff* and bold in *troff*.

1.3 Overall Structure of a Document

The input for a document that is to be formatted with PWB/MM possesses four major segments, any of which may be omitted; if present, they *must* occur in the following order:

- *Parameter-setting*—This segment sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for headings and lists, page headers and footers [9], and many other properties of the document. Also, the user can add macros or redefine existing ones. This segment can be omitted entirely if one is satisfied with default values; it produces no actual output, but only performs the setup for the rest of the document.
- *Beginning*—This segment includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body*—This segment is the actual text of the document. It may be as small as a single paragraph, or as large as hundreds of pages. It may have a hierarchy of *headings* up to seven levels deep [4]. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Six additional levels of subordination are provided by a set of *list* macros for automatic numbering, alphabetic sequencing, and "marking" of list items [5]. The body may also contain various types of displays, tables, figures, and footnotes [7, 8].
- *Ending*—This segment contains those items that occur once only, at the end of a document. Included here are signature(s) and lists of notations (e.g., "copy to" lists) [6.12]. Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document [10].

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed in several different ways depending on the document type, there is a uniform way of typing it in.

1.4 Definitions

The term *formatter* refers to either of the text-formatting programs *nroff* and *troff*.

Requests are built-in commands recognized by the formatters. Although one seldom needs to use these requests directly [3.9], this document contains references to some of them. Full details are given in [9]. For example, the request:

```
.sp
```

inserts a blank line in the output.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. PWB/MM supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by *requests* and *macros*. A string can be given a value via the .ds (define string) request, and its value can be obtained by referencing its name, preceded by "*" (for 1-character names) or "*(" (for 2-character names). For instance, the string *DT* in PWB/MM normally contains the current date, so that the *input* line:

```
Today is \*(DT.
```

may result in the following *output:*

Today is October 31, 1977.

The current date can be replaced, e.g.:

.ds DT 01/01/76

or by invoking a macro designed for that purpose {6.7.1}.

Number registers fill the role of integer variables. They are used for flags, for arithmetic, and for automatic numbering. A register can be given a value using a .nr request, and be referenced by preceding its name by "\n" (for 1-character names) or "\n(" (for 2-character names). For example, the following sets the value of the register *d* to 1 more than that of the register *dd*:

.nr d 1+\n(dd

See {13.1} regarding naming conventions for requests, macros, strings, and number registers.

1.5 Prerequisites and Further Reading

1.5.1 Prerequisites. We assume familiarity with UNIX at the level given in [3] and [4]. Some familiarity with the request summary in [9] is helpful.

1.5.2 Further Reading. [9] provides detailed descriptions of formatter capabilities, while [5] provides a general overview. See [6] (and possibly [7]) for instructions on formatting mathematical expressions. See *tbl*(I) and [11] for instructions on formatting tabular data.

Examples of formatted documents and of their respective input, as well as a quick reference to the material in this manual are given in [8].

2. INVOKING THE MACROS

This section tells how to access PWB/MM, shows PWB/UNIX command lines appropriate for various output devices, and describes command-line flags for PWB/MM. Note that file names, program names, and typical command sequences apply only to PWB/UNIX; different names and command lines may have to be used on other systems.

2.1 The mm Command

The *mm*(I) command can be used to print documents using *nroff* and PWB/MM; this command invokes *nroff* with the -mm flag {2.2}. It has options to specify preprocessing by *tbl*(I) and/or by *neqn*(I), and for postprocessing by various output filters. Any arguments or flags that are not recognized by *mm*(I), e.g. -rC3, are passed to *nroff* or to PWB/MM, as appropriate. The options, which can occur in any order but *must* appear before the file names, are:

-e	*neqn*(I) is to be invoked.
-t	*tbl*(I) is to be invoked.
-c	*col*(I) is to be invoked.
-12	need 12-pitch mode. Be sure that the pitch switch on the terminal is set to 12.
-300	output is to a DASI300 terminal. This is the *default* terminal type.
-hp	output is to a HP264x.
-450	output is to a DASI450.
-tn	output is to a GE TermiNet 300.
-tn300	output is to a GE TermiNet 300.
-ti	output is to a Texas Instrument 700 series terminal.
-37	output is to a TELETYPE® Model 37.

2.2 The -mm Flag

The PWB/MM package can also be invoked by including the **-mm** flag as an argument to the formatter. It causes the file /usr/lib/tmac.m to be read and processed before any other files. This action defines the PWB/MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the files of input text.

2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in {2.4} and in [9]).

- Text without tables or equations:

  mm [options] filename ...
  or **nroff** [options] **-mm** filename ...
  or **troff** [options] **-mm** filename ...

- Text with tables:

  mm -t [options] filename ...
  or **tbl** filename ... | **nroff** [options] **-mm -**
  or **tbl** filename ... | **troff** [options] **-mm -**

- Text with equations:

  mm -e [options] filename ...
  or **neqn** filename ... | **nroff** [options] **-mm -**
  or **eqn** filename ... | **troff** [options] **-mm -**

- Text with both tables and equations:

  mm -t -e [options] filename ...
  or **tbl** filename ... | **neqn** | **nroff** [options] **-mm -**
  or **tbl** filename ... | **eqn** | **troff** [options] **-mm -**

When formatting a document with *nroff*, the output should normally be processed for a specific type of terminal, because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly-used terminal types and the command lines appropriate for them are given below. See {2.4} as well as *gsi*(1), *hp*(1), *col*(1), and *terminals*(VII) for further information.

- DASI300 (GSI300/DTC300) in 10-pitch, 6 lines/inch mode and a line length of 65 characters:

  mm filename ...
  or **nroff -T300 -h -mm** filename ...

- DASI300 (GSI300/DTC300) in 12-pitch, 6 lines/inch mode and a line length of 80—rather than 65—characters:

  mm -12 filename ...
  or **nroff -T300-12 -rW80 -rO3 -h -mm** filename ...

  or, equivalently (and more succinctly):

  nroff -T300-12 -rT1 -h -mm filename ...

- DASI450 in 10-pitch, 6 lines/inch mode:

  mm -450 filename ...
  or **nroff -T450 -h -mm** filename ...

- DASI450 in 12-pitch, 6 lines/inch mode:

  mm -450 -12 filename ...
  or **nroff -T450-12 -rW80 -rO3 -h -mm** filename ...
  or **nroff -T450-12 -rT1 -h -mm** filename ...

- Hewlett-Packard HP264x CRT family:

  mm -hp filename ...
  or **nroff -h -mm** filename ... | **hp**

- Any terminal incapable of reverse paper motion (GE TermiNet, Texas Instruments 700 series, etc.):

  mm -tn filename ...
  or **nroff -mm** filename ... | **col**

- Versatec printer (see *vp*(1) for additional details):

    vp [vp-options] "**mm -rT2** -c filename ..."
  or **vp** [vp-options] "**nroff -rT2 -mm** filename ... | **col**"

Of course, *tbl*(1) and *eqn*(1)/*neqn*(1), if needed, must be invoked as shown in the command line prototypes at the beginning of this section.

If two-column processing {11.4} is used with *nroff*, the -c option must be specified to *mm*(1), or the *nroff* output postprocessed by *col*(1). In the latter case, the -T37 terminal type must be specified to *nroff*, the -h option must *not* be specified, and the output of *col*(1) must be processed by the appropriate terminal filter (e.g., *gsi*(1)); *mm*(1) with the -c option handles all this automatically.

2.4 Parameters that Can Be Set from the Command Line

Number registers are commonly used within PWB/MM to hold parameter values that control various aspects of output style. Many of these can be changed within the text files via .nr requests. In addition, some of these registers can be set from the command line itself, a useful feature for those parameters that should *not* be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register *P*—see below) *must* be set on the command line (or before the PWB/MM macro definitions are processed) and their meanings are:

-rA1 has the effect of invoking the .AF macro without an argument {6.7.2}.

-rB*n* defines the macros for the cover sheet and the table of contents. If *n* is 1, table-of-contents processing is enabled. If *n* is 2, then cover-sheet processing will occur. If *n* is 3, both will occur. That is, *B* having a value greater than 0 *defines* the .TC {10.1} and/or .CS {10.2} macros. Note that to have any effect, these macros must also be *invoked*.

-rC*n* *n* sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page. See {9.5}.
n = 1 for OFFICIAL FILE COPY.
n = 2 for DATE FILE COPY.
n = 3 for DRAFT.

-rD1 sets *debug mode*. This flag requests the formatter to attempt to continue processing even if PWB/MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2, 11.3}.

-rL*k* sets the length of the physical page to *k* lines.[1] The default value is 66 lines per page. This parameter is used for obtaining 8 lines-per-inch output on 12-pitch terminals, or when directing output to a Versatec printer.

-rN*n* specifies the page numbering style. When *n* is 0 (default), all pages get the (prevailing) header {9.2}. When *n* is 1, the page header replaces the footer on page 1 only. When *n* is 2, the page header is omitted from page 1. When *n* is 3, "section-page" numbering {4.5} occurs.

n	*Page 1*	*Pages 2 ff.*
0	header	header
1	header replaces footer	header
2	no header	header
3	"section-page" as *footer*	

The contents of the prevailing header and footer do *not* depend on of the value of the number register *N*; *N* only controls whether and where the header (and, for *N*= 3, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null {9.2, 9.5}, the value of *N* is irrelevant.

-rO*k* offsets output *k* spaces to the right.[1] It is helpful for adjusting output positioning on some terminals. NOTE: The register name is the capital letter "O", *not* the digit zero (0).

-rP*n* specifies that the pages of the document are to be numbered starting with *n*. This register may also be set via a .nr request in the input text.

1. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

-rS*n* sets the point size and vertical spacing for the document. The default *n* is 10, i.e., 10-point type on 12-point leading (vertical spacing), giving 6 lines per inch {11.8}. This parameter applies to *troff* only.

-rT*n* provides register settings for certain devices. If *n* is 1, then the line length and page offset are set for output directed to a DASI300 or DASI450 in 12-pitch, 6 lines/inch mode, i.e., they are set to 80 and 3, respectively. Setting *n* to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for *n* is 0. This parameter applies to *nroff* only.

-rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}. This parameter applies to *nroff* only.

-rW*k* page width (i.e., line length and title length) is set to *k*.[2] This can be used to change the page width from the default value of 65 characters (6.5 inches).

2.5 Omission of -mm

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac.m
remainder of text
```

In this case, one must *not* use the **-mm** flag (nor the *mm*(1) command); the .so request has the equivalent effect, but the registers in {2.4} must be initialized *before* the .so request, because their values are meaningful only if set before the macro definitions are processed. When using this method, it is best to "lock" into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.nr B 1
.so /usr/lib/tmac.m
.H 1 "INTRODUCTION"
   .
   .
   .
```

specifies, for *nroff*, a line length of 80, a page offset of 10, "section-page" numbering, and table of contents processing.

3. FORMATTING CONCEPTS

3.1 Basic Terms

The normal action of the formatters is to *fill* output lines from one or more input lines. The output lines may be *justified* so that both the left and right margins are aligned. As the lines are being filled, words are hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (see .SA {11.2}, *Hy* {3.4}, and the formatter .nf and .fi requests [9]). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause the filling of the current output line to cease, the line (of whatever length) to be printed, and the subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the PWB/MM macros cause a break.

While formatter requests can be used with PWB/MM, one must fully understand the consequences and side-effects that each such request might have. Actually, there is little need to use formatter requests; the macros described here should be used in most cases because:

---

2. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

— it is much easier to control (and change at any later point in time) the overall style of the document.

— complicated facilities (such as footnotes or tables of contents) can be obtained with ease.

— the user is insulated from the peculiarities of the formatter language.

A good rule is to use formatter requests only when absolutely necessary {3.9}.

In order to make it easy to revise the input text at a later time, input lines should be kept short and should be broken at the end of clauses; each new full *sentence must* begin on a new line.

3.2 Arguments and Double Quotes

For any macro call, a *null argument* is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "". Note that *omitting* an argument is *not* the same as supplying a *null argument* (for example, see the .MT macro in {6.6}). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces *must* be enclosed in double quotes (").[3] Otherwise, it will be treated as several separate arguments.

Double quotes (") are *not* permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If you must, use two grave accents (``) and/or two acute accents ('') instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times; for example, headings are first printed in the text and may be (re)printed in the table of contents.

3.3 Unpaddable Spaces

When output lines are *justified* to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this.

First, one may type a backslash followed by a space ("\ "). This pair of characters directly generates an *unpaddable space*. Second, one may sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily "recovered" by inserting:

```
.tr ~~
```

before the place where it is needed. Its previous usage is restored by repeating the ".tr ~", but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is *not* recommended for documents in which the tilde is used within equations.

3.4 Hyphenation

The formatters (and, therefore, PWB/MM) will automatically hyphenate words, if need be. However, the user may specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation indicator, or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the two-character sequence "\%") appears at the beginning or end of a word, the word is *not* hyphenated. Alternatively, it can be used to indicate legal hyphenation point(s) inside a word. In any case, *all* occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator:

```
.HC [hyphenation-indicator]
```

3. A double quote (") is a *single* character that must not be confused with two apostrophes or acute accents (''), or with two grave accents (``).

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

```
.HC ^
```

Note that any word containing hyphens or dashes—also known as *em* dashes—will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, *even if the formatter hyphenation function is turned off.*

Hyphenation can be turned off in the body of the text by specifying:

```
.nr Hy 0
```

once at the beginning of the document. For hyphenation control within footnote text and across pages, see [8.3].

The user may supply, via the .hw request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word "printout," one may specify:

```
.hw print-out
```

3.5 Tabs

The macros .MT [6.6], .TC [10.1], and .CS [10.2] use the formatter .ta request to set tab stops, and then restore the *default* values[4] of tab settings. Thus, setting tabs to other than the default values is the user's responsibility.

Note that a tab character is always interpreted with respect to its position on the *input line*, rather than its position on the output line. In general, tab characters should appear only on lines processed in "no-fill" mode [3.1].

Also note that *tbl*(1) [7.3] changes tab stops, but does *not* restore the default tab settings.

3.6 Special Use of the BEL Character

The non-printing character BEL is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers [9], headings [4], and list marks [5]. Users who include BEL characters in their input text (especially in arguments to macros) will receive mangled output.

3.7 Bullets

A bullet (•) is often obtained on a typewriter terminal by using an "o" overstruck by a "+". For compatibility with *troff*, a bullet string is provided by PWB/MM. Rather than overstriking, use the sequence:

```
\*(BU
```

wherever a bullet is desired. Note that the bullet list (.BL) macros [5.3.3.2] use this string to automatically generate the bullets for the list items.

3.8 Dashes, Minus Signs, and Hyphens

Troff has distinct graphics for a dash, a minus sign, and a hyphen, while *nroff* does not. Those who intend to use *nroff* only may use the minus sign ("-") for all three.

Those who wish mainly to use *troff* should follow the escape conventions of [9].

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

Dash — Type " -- " for each text dash. These can be left alone for *nroff*, and later globally translated for *troff* to "\(em", namely an em dash (—). Note that the dash list (.DL) macros [5.3.3.3] automatically generate the em dashes for the list items.

4. Every eight characters in *nroff*; every ½ inch in *troff*.

Hyphen Type "-" and use as is for both formatters. *Nroff* will print it as is, and *troff* will print a true hyphen.

Minus Type "\-" for a true minus sign, regardless of formatter. *Nroff* will effectively ignore the "\", while *troff* will print a true minus sign.

3.9 Use of Formatter Requests

Most formatter requests [9] should *not* be used with PWB/MM because PWB/MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests [3.1]. However, some formatter requests *are* useful with PWB/MM, namely:

.af	.br	.ce	.de	.ds	.fi	.hw	.ls	.nf	.nr
.nx	.rm	.rr	.rs	.so	.sp	.ta	.ti	.tl	.tr

The .fp, .lg, and .ss requests are also sometimes useful for *troff*. Use of other requests without fully understanding their implications very often leads to disaster.

4. PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in [5].

4.1 Paragraphs

.P [type]
one or more lines of text.

This macro is used to begin two kinds of paragraphs. In a *left-justified* paragraph, the first line begins at the left margin, while in an *indented* paragraph, it is indented five spaces (see below).

A document possesses a *default paragraph style* obtained by specifying ".P" before each paragraph that does *not* follow a heading [4.2]. The default style is controlled by the register *Pt*. The initial value of *Pt* is 2, which provides indented paragraphs *except* after headings, lists, and displays, in which case they are left-justified. All paragraphs can be forced to be left-justified by inserting the following at the beginning of the document:

```
.nr Pt 0
```

All paragraphs can be forced to be indented by inserting:

```
.nr Pt 1
```

at the beginning of the document.

The amount a paragraph is indented is contained in the register *Pi*, whose default value is 5. To indent paragraphs by, say, 10 spaces, insert:

```
.nr Pi 10
```

at the beginning of the document. Of course, both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

☛ *Values that specify indentation must be* unscaled *and are treated as "character positions," i.e., as a number of* ens. *In* troff, *an* en *is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In* nroff, *an* en *is equal to the width of a character.*

Regardless of the value of *Pt*, an *individual* paragraph can be forced to be left-justified or indented. ".P 0" always forces left justification; ".P 1" always causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a *list*, the indent (if any) of the paragraph is added to the current list indent [5].

4.2 Numbered Headings

.H level [heading-text]
zero or more lines of text

The .H macro provides seven levels of numbered headings, as illustrated by this document. Level 1 is the most major or highest; level 7 the lowest.

☛ *There is no need for a .P macro after a .H (or .HU {4.3}), because the .H macro also performs the function of the .P macro. In fact, if a .P follows a .H, the user loses much of the flexibility provided by the .H mechanism* {4.2.2.2}.

4.2.1 Normal Appearance. The normal appearance of headings is as shown in this document. The effect of .H varies according to the *level* argument. First-level headings are *preceded* by two blank lines (one vertical space); all others are *preceded* by one blank line (½ a vertical space).

.H 1 heading-text — gives an underlined (bold) heading *followed* by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.

.H 2 heading-text — yields an underlined (bold) heading followed by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.

.H *n* heading-text — for $3 \leqslant n \leqslant 7$, produces an underlined (italic) heading followed by two spaces. The following text appears on the same line, i.e., these are *run-in* headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a .H macro call.

Here are the first few .H calls of {4}:

```
.H 1 "PARAGRAPHS AND HEADINGS"
.H 2 "Paragraphs"
.H 2 "Numbered Headings"
.H 3 "Normal Appearance."
.H 3 "Altering Appearance of Headings."
.H 4 "Pre-Spacing and Page Ejection."
.H 4 "Spacing After Headings."
.H 4 "Centered Headings."
.H 4 "Bold, Italic, and Underlined Headings."
.H 5 "Control by Level."
```

4.2.2 Altering Appearance of Headings. Users satisfied with the default appearance of headings may skip to {4.3}. One can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines, rather than being distributed throughout the document.

4.2.2.1 Pre-Spacing and Page Ejection. A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line (½ a vertical space). If a multi-line heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents may be made more manageable if each section starts on a new page. Setting *Ej* to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to *Ej*.

4.2.2.2 Spacing After Headings. Three registers control the appearance of text immediately following a .H call. They are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break {3.1} occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line (½ a vertical space) is inserted after the heading. Defaults for *Hb* and *Hs* are 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the

heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any *stand-alone* heading, i.e., a heading not run into the following text, the alignment of the next line of output is controlled by the register *Hi*. If *Hi* is 0, text is left-justified. If *Hi* is 1 (the *default* value), the text is indented according to the paragraph type as specified by the register *Pt* {4.1}. Finally, if *Hi* is 2, text is indented to line up with the first word of the heading itself, so that the heading number stands out more clearly. Note that this feature is defeated if a .P macro follows the .H or .HU macro {4.2}.

For example, to cause a blank line (½ a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the top of the document:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

4.2.2.3 Centered Headings. The register *Hc* can be used to obtain centered headings. A heading is centered if its level is less than or equal to *Hc, and* if it is also stand-alone {4.2.2.2}. *Hc* is 0 initially (no centered headings).

4.2.2.4 Bold, Italic, and Underlined Headings.

4.2.2.4.1 Control by Level. Any heading that is underlined by *nroff* is made bold or italic by *troff*. The string *HF* (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for *HF* are:

Formatter	*HF Code*			*Default*
	1	*2*	*3*	*HF*
nroff	no underline	underline	underline	3 3 2 2 2 2 2
troff	roman	italic	bold	3 3 2 2 2 2 2

Thus, all levels are underlined in *nroff*; in *troff*, levels 1 and 2 are bold, levels 3 through 7 are italic. The user may reset HF as desired. Any value omitted from the right end of the list is taken to be 1. For example, the following would result in five underlined (bold) levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

4.2.2.4.2 Nroff Underlining Style. Nroff can underline in two ways. The normal style (.ul request) is to underline only letters and digits. The continuous style (.cu request) underlines all characters, including spaces. By default, PWB/MM attempts to use the continuous style on any heading that is to be underlined, is *not* run-in, and is short enough to fit on a single line. If a heading is to be underlined, but is either run-in or is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the -rU1 flag when invoking *nroff* {2.4}.

4.2.2.5 Marking Styles—Numerals and Concatenation.

```
.HM [arg1] ... [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The .HM macro (heading mark style) allows this choice to be overridden, thus providing "outline" and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

Value	*Interpretation*
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Upper-case alphabetic
a	Lower-case alphabetic
I	Upper-case Roman
i	Lower-case Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, set the register *Ht* (heading-mark type) to 1.

For example, a commonly-used "outline" style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

4.3 Unnumbered Headings

.HU heading-text

.HU is a special case of .H; it is handled in the same way as .H, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when .H and .HU calls are intermixed, each .HU heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between:

.HU heading-text

and

.H 2 heading-text

is the printing of the heading mark for the latter. Both have the effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

.HU can be especially helpful in setting up Appendices and other sections that may not fit well into the numbering scheme of the main body of a document {13.2.1}.

4.4 Headings and the Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following three things:

- specifying in the register *Cl* what level headings are to be saved;
- invoking the .TC macro {10.1} at the end of the document;
- and specifying -rB*n* {2.4} on the command line.

Any heading whose level is less than or equal to the value of the register *Cl* (contents level) is saved and later displayed in the table of contents. The default value for *Cl* is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the "Out of temp file space" diagnostic {Appendix E} will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5 First-Level Headings and the Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the form "section-page," where *section* is the number of the current first-level heading. This page numbering style can be achieved by specifying the flag -rN3 on the command line {9.9}. As a side effect, this also has the effect of setting *Ej* to 1, i.e., each section

begins on a new page. In this style, the page number is printed at the *bottom* of the page, so that the correct section number is printed.

4.6 User Exit Macros •

☛ *This section is intended only for users who are accustomed to writing formatter macros.*

.HX dlevel rlevel heading-text
.HZ dlevel rlevel heading-text

The .HX and .HZ macros are the means by which the user obtains a final level of control over the previously-described heading mechanism. PWB/MM does not define .HX and .HZ; they are intended to be defined by the user. The .H macro invokes .HX shortly before the actual heading text is printed; it calls .HZ as its last action. All the default actions occur if these macros are not defined. If the .HX or .HZ (or both) are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the .HU macro is actually a special case of the .H macro.

If the user originally invoked the .H macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the .H invocation. If the user originally invoked the .HU macro {4.3}, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time .H calls .HX, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank line(s) (vertical space) to precede the heading {4.2.2.1}, and accumulated the "heading mark", i.e., the string of digits, letters, and periods needed for a numbered heading. When .HX is called, all user-accessible registers and strings can be referenced, as well as the following:

string }0 — If *rlevel* is non-zero, this string contains the "heading mark." Two unpaddable spaces (to separate the *mark* from the *heading*) have been appended to this string. If *rlevel* is 0, this string is null.

register ;0 — This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (½ a vertical space) is to follow the heading.

string }2 — If register ;0 is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) *heading* from the following *text*. If register ;0 is non-zero, this string is null.

register ;3 — This register contains an adjustment factor for a .ne request issued before the heading is actually printed. On entry to .HX, it has the value 3 if *dlevel* equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the register ;0 taken as blank lines (halves of vertical space) plus the contents of register ;3 as blank lines (halves of vertical space) plus the number of lines of the heading.

The user may alter the values of *}0*, *}2*, and *;3* within .HX as desired. The following are examples of actions that might be performed by defining .HX to include the lines shown:

Change first-level heading mark from format *n.* to *n.*0:
.if \\$1=1 .ds }0 \\n(H1.0\□\□ (□ stands for a space)

Separate run-in heading from the text with a period and two unpaddable spaces:
.if \\n(;0=0 .ds }2 .\□\□

Assure that at least 15 lines are left on the page before printing a first-level heading:
.if \\$1=1 .nr ;3 15-\\n(;0

Add 3 additional blank lines before each first-level heading:
.if \\$1=1 .sp 3

If temporary string or macro names are used within .HX, care must be taken in the choice of their names {13.1}.

.HZ is called at the end of .H to permit user-controlled actions after the heading is produced. For example, in a large document, sections may correspond to chapters of a book, and the user may want to reset counters for footnotes, figures, tables, etc. Another use might be to change a page header or footer. For example:

```
.de HZ
.if \\$1=1 \{.nr :p 0 \" footnotes
.      nr Fg 0 \" figures
.      nr Tb 0 \" tables
.      nr Ec 0 \" equations
.      PF "''Section \\$3''"\}
..
```

4.7 Hints for Large Documents

A large document is often organized for convenience into one file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register *H1* to 1 less than the number of the section just *before* the corresponding ".H 1" call. For example, at the beginning of section 5, insert:

```
.nr H1 4
```

☛ *This is a dangerous practice: it defeats the automatic (re)numbering of sections when sections are added or deleted. Remove such lines as soon as possible.*

5. LISTS

This section describes many different kinds of lists: automatically-numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, e.g., with terms or phrases to be defined.

5.1 Basic Approach

In order to avoid repetitive typing of arguments to describe the appearance of items in a list, PWB/MM provides a convenient way to specify lists. All lists are composed of the following parts:

- A *list-initialization* macro that controls the appearance of the list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing.
- One or more *List Item* (.LI) macros, each followed by the actual text of the corresponding list item.
- The *List End* (.LE) macro that terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list-initialization macro saves the previous list status (indentation, marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only once at the beginning of that list. In addition, by building on the existing structure, users may create their own customized sets of list macros with relatively little effort {5.4, Appendix A, Appendix B}.

5.2 Sample Nested Lists

The input for several lists and the corresponding output are shown below. The .AL and .DL macro calls {5.3.3} contained therein are examples of the *list-initialization* macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a "plus" as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized item, B.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph appears at the left margin.
```

Output:

A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

   1. This is a numbered item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

      — This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

   + — This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

   2. This is numbered item 2.

B. This is another alphabetized item, B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph appears at the left margin.

5.3 Basic List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists. Each list-initialization macro is covered in {5.3.3}.

5.3.1 List Item.

```
.LI [mark] [1]
one or more lines of text that make up the list item.
```

The .LI macro is used with all lists. It normally causes the output of a single blank line (½ a vertical space) before its item, although this may be suppressed. If no arguments are given, it labels its item with the *current mark*, which is specified by the most recent list-initialization macro. If a single argument is given to .LI, that argument is output *instead of* the current mark. If two arguments are given, the first argument becomes a *prefix* to the current mark, thus allowing the user to emphasize one or more items in a list. One unpaddable space is inserted between the prefix and the mark. For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a "plus."
.LI + xxx
But this uses "plus" as prefix to the bullet.
.LE
```

yields:

- This is a simple bullet item.

+ This replaces the bullet with a "plus."

+ • But this uses "plus" as prefix to the bullet.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

If the *current mark* (in the *current list*) is a null string, and the first argument of .LI is omitted or null, the resulting effect is that of a *hanging indent*, i.e., the first line of the following text is "outdented," starting at the same place where the *mark* would have started {5.3.3.6}.

5.3.2 List End.

.LE [1]

List End restores the state of the list back to that existing just before the most recent list-initialization macro call. If the optional argument is given, the .LE outputs a blank line (½ a vertical space). This option should generally be used only when the .LE is followed by running text, but not when followed by a macro that produces blank lines of its own, such as .P, .H, or .LI.

.H and .HU automatically clear all list information, so one may legally omit the .LE(s) that would normally occur just before either of these macros. Such a practice is *not* recommended, however, because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.3.3 List Initialization Macros. The following are the various list-initialization macros. They are actually implemented as calls to the more basic .LB macro {5.4}.

5.3.3.1 Automatically-Numbered or Alphabetized Lists.

.AL [type] [text-indent] [1]

The .AL macro is used to begin sequentially-numbered or alphabetized lists. If there are no arguments, the list is numbered, and text is indented *Li* (initially 5)[5] spaces from the indent in force when the .AL is called, thus leaving room for two digits, a period, and two spaces before the text.

The *type* argument may be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, I, or i {4.2.2.5}.[6] If *type* is omitted or null, then "1" is assumed. If *text-indent* is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* is null, then the value of *Li* will be used.

5. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

6. Note that the "0001" format is *not* permitted.

If the third argument is given, a blank line (½ a vertical space) will *not* separate the items in the list. A blank line (½ a vertical space) will occur before the first item, however.

5.3.3.2 Bullet List.

.BL [text-indent] [1]

.BL begins a bullet list, in which each item is marked by a bullet (•) followed by one space. If *text-indent* is non-null, it overrides the default indentation—the amount of paragraph indentation as given in the register *Pi* {4.1}.[7]

If a second argument is specified, no blank lines will separate the items in the list.

5.3.3.3 Dash List.

.DL [text-indent] [1]

.DL is identical to .BL, except that a dash is used instead of a bullet.

5.3.3.4 Marked List.

.ML mark [text-indent] [1]

.ML is much like .BL and .DL, but expects the user to specify an arbitrary mark, which may consist of more than a single character. Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*. If the third argument is specified, no blank lines will separate the items in the list.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

5.3.3.5 Reference List.

.RL [text-indent] [1]

A .RL call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([]). *Text-indent* may be supplied, as for .AL. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list. The list of references {14} was produced using the .RL macro.

5.3.3.6 Variable-Item List.

.VL text-indent [mark-indent] [1]

When a list begins with a .VL, there is effectively no *current mark;* it is expected that each .LI will provide its own mark. This form is typically used to display definitions of terms or phrases. *Mark-indent* gives the number of spaces from the current indent to the beginning of the *mark*, and it defaults to 0 if omitted or null. *Text-indent* gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of .VL usage:

7. So that, in the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

```
.tr ~
.VL 20 2
.LI mark~1
Here is a description of mark 1;
"mark 1" of the .LI line contains a tilde translated to an unpaddable space in order
to avoid extra spaces between
"mark" and "1" (3.3).
.LI second~mark
This is the second mark, also using a tilde translated to an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark; one space separates the mark
from the text.
.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

mark 1 Here is a description of mark 1; "mark 1" of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between "mark" and "1" (3.3).

second mark This is the second mark, also using a tilde translated to an unpaddable space.

third mark longer than indent: This item shows the effect of a long mark; one space separates the mark from the text.

This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a *hanging indent* would have been produced. A *hanging indent* is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

yields:

Here is some text to show a hanging indent. The first line of text is at the left margin. The second is indented 10 spaces.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* (3.3).

5.4 List-Begin Macro and Customized Lists •

.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

The list-initialization macros described above suffice for almost all cases. However, if necessary, one may obtain more control over the layout of lists by using the basic list-begin macro .LB, which is also used by all the other list-initialization macros (Appendix A). Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register *Li* for automatic lists and from the register *Pi* for bullet and dash lists.

The combination of *mark-indent* and *pad* determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent, and

ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent).[8] Within the mark area, the mark is *left-justified* if *pad* is 0. If *pad* is greater than 0, say *n*, then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively *right-justified pad* spaces immediately to the left of the text.

Type and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done, and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.3.3.1}. That is:

Type	*Mark*	*Result*
0	omitted	hanging indent
0	*string*	*string* is the mark
>0	omitted	arabic numbering
>0	one of:	automatic numbering or
	1, A, a, I, i	alphabetic sequencing

Each non-zero value of *type* from 1 to 6 selects a different way of displaying the items. The following table shows the output appearance for each value of *type:*

Type	*Appearance*
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

where *x* is the generated number or letter.

☛ *The* mark *must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified* {3.3}.

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each .LI macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the .LI macro issues a .ne request for two lines just before printing the mark.

LB-space, the number of blank lines (½ a vertical space) to be output by .LB itself, defaults to 0 if omitted.

There are three reasonable combinations of *LI-space* and *LB-space*. The normal case is to set *LI-space* to 1 and *LB-space* to 0, yielding one blank line *before* each item in the list; such a list is usually terminated with a ".LE 1" to end the list with a blank line. In the second case, for a more compact list, set *LI-space* to 0 and *LB-space* to 1, and, again, use ".LE 1" at the end of the list. The result is a list with one blank line before and after it. If you set both *LI-space* and *LB-space* to 0, and use ".LE" to end the list, a list without *any* blank lines will result.

Appendix A shows the definitions of the list-initialization macros {5.3.3} in terms of the .LB macro. Appendix B illustrates how the user can build upon those macros to obtain other kinds of lists.

6. MEMORANDUM AND RELEASED PAPER STYLES

One use of PWB/MM is for the preparation of memoranda and released papers, which have special requirements for the first page and for the cover sheet. The information needed for the memorandum or released paper (title, author, date, case numbers, etc.) is entered in the same way for *both* styles; an argument to one macro indicates which style is being used. The following sections describe the macros used to provide this data. The required order is shown in {6.9}.

8. The *mark-indent* argument is typically 0.

If neither the memorandum nor released-paper style is desired, the macros described below should be omitted from the input text. If these macros are omitted, the first page will simply have the page header {9} followed by the body of the document.

6.1 Title

.TL [charging-case] [filing-case]
one or more lines of title text

The arguments to the .TL macro are the charging case number(s) and filing case number(s).[9] The title of the memorandum or paper follows the .TL macro and is processed in fill mode {3.1}. Multiple charging case numbers are entered as "sub-arguments" by separating each from the previous with a comma and a space, and enclosing the *entire* argument within double quotes. Multiple filing case numbers are entered similarly. For example:

.TL "12345, 67890" 987654321
On the construction of a table
of all even prime numbers

The .br request may be used to break the title into several lines.

On output, the title appears after the word "subject" in the memorandum style. In the released-paper style, the title is centered and underlined (bold).

6.2 Author(s)

.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

The .AU macro receives as arguments information that describes an author. If any argument contains blanks, it must be enclosed within double quotes. The first six arguments must appear in the order given (a separate .AU macro is required for each author). For example:

.AU "J. J. Jones" JJJ PY 9876 5432 1Z-234

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. The printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9, if present, will follow this "normal" author information, each on a separate line. Certain organizations have their own numbering schemes for memoranda, engineer's notes, etc. These numbers are printed after the author's name. This can be done by providing more than six arguments to the .AU macro, e.g.:

.AU "S. P. Lename" SPL IH 9988 7766 5H-444 3322.11AB

The name, initials, location, and department are also used in the Signature Block {6.11.1}. The author information in the "from" portion, as well as the names and initials in the Signature Block will appear in the same order as the .AU macros.

The names of the authors in the released-paper style are centered below the title. After the name of the last author, "Bell Laboratories" and the location are centered. For the case of authors from different locations, see {6.8}.

6.3 TM Number(s)

.TM [number] ...

If the memorandum is a Technical Memorandum, the TM numbers are supplied via the .TM macro. Up to nine numbers may be specified. Example:

.TM 7654321 77777777

This macro call is ignored in the released-paper and external-letter styles {6.6}.

---

9. The "charging case" is the case number to which time was charged for the development of the project described in the memorandum. The "filing case" is a number under which the memorandum is to be filed.

6.4 Abstract

.AS [arg] [indent]
text of the abstract
.AE

In both the memorandum and released-paper styles, the text of the abstract follows the author information and is preceded by the centered and underlined (italic) word "ABSTRACT."

The .AS (abstract start) and .AE (abstract end) macros bracket the (optional) abstract. The first argument to .AS controls the printing of the abstract. If it is 0 or null, the abstract is printed on the first page of the document, immediately following the author information, and is also saved for the cover sheet. If the first argument is 1, the abstract is saved and printed only on the cover sheet. The margins of the abstract are indented on the left and right by five spaces. The amount of indentation can be changed by specifying the desired indentation as the second argument.[10]

Note that headings {4.2, 4.3}, displays {7}, and footnotes {8} are *not* (as yet) permitted within an abstract.

6.5 Other Keywords

.OK [keyword] ...

Topical keywords should be specified on a Technical Memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.6 Memorandum Types

.MT [type] [1]

The .MT macro controls the format of the top part of the first page of a memorandum or of a released paper, as well as the format of the cover sheets. Legal codes for *type* and the corresponding values are:

Code	*Value*
.MT ""	no memorandum type is printed
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER'S NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Released-Paper style
.MT 5	External-Letter style
.MT "*string*"	*string*

If *type* indicates a memorandum style, then *value* will be printed after the last line of author information or after the last line of the abstract, if one appears on the first page. If *type* is longer than one character, then it, itself, will be printed. For example:

.MT "Technical Note #5"

A simple letter is produced by calling .MT with a null (but *not* omitted!) or zero argument.

The second argument to .MT is used only if the first argument is 4 (i.e., for the released-paper style) as explained in {6.8}.

In the external-letter style (.MT 5), only the date is printed in the upper right corner of the first page. It is expected that preprinted stationery will be used, providing the author's company logotype and address.

10. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

6.7 Date and Format Changes

6.7.1 Changing the Date. By default, the current date appears in the "date" part of a memorandum. This can be overridden by using:

.ND new-date

The .ND macro alters the value of the string *DT*, which is initially set to the current date.

6.7.2 Alternate First-Page Format. One can specify that the words "subject," "date," and "from" (in the memorandum style) be omitted and that an alternate company name be used:

.AF [company-name]

If an argument is given, it replaces "Bell Laboratories", without affecting the other headings. If the argument is *null*, "Bell Laboratories" is suppressed; in this case, extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp. .AF with *no* argument suppresses "Bell Laboratories" and the "Subject/Date/From" *headings*, thus allowing output on preprinted stationery.

The only .AF option appropriate for *troff* is to specify an argument to replace "Bell Laboratories" with another name.

6.8 Released-Paper Style

The released-paper style is obtained by specifying:

.MT 4 [1]

This results in a centered, underlined (bold) title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless .AF {6.7.2} specifies a different company). If the optional second argument to .MT is given, then the name of each author is followed by the respective company name and location. The abstract, if present, follows the author information.

Information necessary for the memorandum style but not for the released-paper style is ignored.

If the released-paper style is utilized, most BTL location codes[11] are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, *following* the .MT macro, the user may re-use these string names. In addition, the macros described in {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate .AF *before* each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.AU "F. Swatter"
.AF "Bell Laboratories"
.AU "Sam P. Lename" "" CB
.MT 4 1
```

6.9 Order of Invocation of "Beginning" Macros

The macros described in {6.1-6.7}, *if present*, must be given in the following order:

11. The complete list is: AK, CP, CH, CB, DR, HO, IN, IH, MV, MH, PY, RR, RD, WV, and WH.

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE
.OK [keyword] ...
.MT [type] [1]
```

The only *required* macros for a memorandum or a released paper are .TL, .AU, and .MT; all the others (and their associated input lines) may be omitted if the features they provide are not needed. Once .MT has been invoked, *none* of the above macros can be re-invoked because they are removed from the table of defined macros to save space.

6.10 Example

The input text for this manual begins as follows:

```
.TL
P\s-3WB/MM\s0\(emProgrammer's Workbench Memorandum Macros
.AU "D. W. Smith" DWS PY ...
.AU "J. R. Mashey" JRM  MH ...
.MT 4 1
```

6.11 Macros for the End of a Memorandum

At the end of a memorandum (but not of a released paper), the signatures of the authors and a list of notations[12] can be requested. The following macros and their input are ignored if the released-paper style is selected.

6.11.1 Signature Block.

```
.SG [arg] [1]
```

.SG prints the author name(s) after the last line of text, aligned with the "Date/From" block. Three blank lines are left above each name for the actual signature. If no argument is given, the line of reference data[13] will *not* appear following the last line.

A non-null first argument is treated as the typist's initials, and is appended to the reference data. Supply a null argument to print reference data with neither the typist's initials nor the preceding hyphen.

If there are several authors and if the second argument is given, then the reference data is placed on the same line as the name of the first author, rather than on the line that has the name of the last author.

The reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the .SG macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

---

12. See [2], pp. 1.12–16

13. The following information is known as reference data: location code, department number, author's initials, and typist's initials, all separated by hyphens. See [2], page 1.11

6.11.2 "Copy to" and Other Notations.

.NS [arg]
zero or more lines of the notation
.NE

After the signature and reference data, many types of notations may follow, such as a list of attachments or "copy to" lists. The various notations are obtained through the .NS macro, which provides for the proper spacing and for breaking the notations across pages, if necessary.

The codes for *arg* and the corresponding notations are:

Code	*Notations*
.NS ""	Copy to
.NS 0	Copy to
.NS	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to
.NS "*string*"	Copy (*string*) to

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to." For example:

.NS "with att. 1 only"

will generate "Copy (with att. 1 only) to" as the notation. More than one notation may be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example:

.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE

would be formatted as:

Atts.
Attachment 1-List of register names
Attachment 2-List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz

6.12 Forcing a One-Page Letter

At times, one would like just a bit more space on the page, forcing the signature or items within notations onto the bottom of the page, so that the letter or memo is just one page in length. This can be accomplished by increasing the page length through the -rL*n* option, e.g. -rL90. This has the effect of

making the formatter believe that the page is 90 lines long and therefore giving it more room than usual to place the signature or the notations. This will *only* work for a *single-page* letter or memo.

7. DISPLAYS

Displays are blocks of text that are to be kept together—not split across pages. PWB/MM provides two styles of displays:[14] a *static* (.DS) style and a *floating* (.DF) style. In the *static* style, the display appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of the page if the display is too big to fit there. In the *floating* style, the display "floats" through the input text to the top of the next page if there is not enough room for it on the current page; thus the input text that *follows* a floating display may *precede* it in the output text. A queue of floating displays is maintained so that their relative order is not disturbed.

By default, a display is processed in no-fill mode and is *not* indented from the existing margin. The user can specify indentation or centering, as well as fill-mode processing.

Displays and footnotes {8} may *never* be nested, in any combination whatsoever. Although lists {5} and paragraphs {4.1} are permitted, no headings (.H or .HU) can occur within displays or footnotes.

7.1 Static Displays

```
.DS [format] [fill]
one or more lines of text
.DE
```

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will *not* indent them from the prevailing indentation. The *format* argument to .DS is an integer with the following meanings:

Code	*Meaning*
""	no indent
0	no indent
1	indent by standard amount
2	center each line

The *fill* argument is also an integer and can have the following meanings:

Code	*Meaning*
""	no-fill mode
0	no-fill mode
1	fill mode

Omitted arguments are taken to be zero.

The standard amount of indentation is taken from the register *Si*, which is initially 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register {4.1}. Even though their initial values are the same, these two registers are independent of one another.

By default, a blank line (½ a vertical space) is placed before and after static and floating displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

7.2 Floating Displays

```
.DF [format] [fill]
one or more lines of text
.DE
```

A floating display is started by the .DF macro and terminated by the .DE macro. The arguments have the same meanings as for .DS {7.1}, except that, for floating displays, indent, no indent, and centering are always calculated with respect to the initial left margin, because the prevailing indent may change

14. Displays are processed in an environment that is different from that of the body of the text (see the .ev request in {9}).

between the time when the formatter first reads the floating display and the time that the display is printed. One blank line (½ a vertical space) *always* occurs both before and after a floating display.

7.3 Tables

```
.DS
.TS
one or more lines of text to be processed by tbl(I)
.TE
.DE
```

The .TS (table start) and .TE (table end) macros make possible the use of the *tbl*(I) processor [11]. They are used *only* to delimit the text to be examined by *tbl*(I). Thus, the display function and the *tbl*(I) delimiting function are independent of one another, in order to permit one to keep together blocks that contain any mixture of tables, equations, filled and unfilled text, and caption lines.

If a particular document does not need this flexibility, it is possible to define .TS and .TE so that they act like .DS and .DE, respectively, and are also recognized by *tbl*(I):

```
.de TS
.Ds "\\$1" "\\$2"
..
.de TE
.DE
..
```

If floating tables are desired, substitute .DF for .DS in the above.

7.4 Equations

```
.DS
.EQ
equation(s)
.EN
.DE
```

The equation setters *eqn*(I) and *neqn*(I) [6,7] expect to use the .EQ (equation start) and .EN (equation end) macros as delimiters in the same way that *tbl*(I) uses .TS and .TE; .EQ and .EN must occur either inside .DS-.DE pairs or else be defined by the user as shown above for the .TS and .TE macros.

☛ *There is an exception to this rule: if .EQ and .EN are used only to specify the delimiters for in-line equations or to specify* eqn/neqn *"defines," .DS and .DE must* not *be used; otherwise extra blank lines will appear in the output.*

7.5 Figure, Table, and Equation Captions

```
.FG  [title] [override] [flag]
.TB  [title] [override] [flag]
.EC  [title] [override] [flag]
```

The .FG (Figure Title), .TB (Table Title), .EC (Equation Caption) macros are normally used inside .DS-.DE pairs to automatically number and title figures, tables, and equations. They use registers *Fg*, *Tb*, and *Ec*, respectively.[15] As an example, the call:

```
.FG "This is an illustration"
```

yields:

Figure 1. This is an illustration

.TB replaces "**Figure**" by "**TABLE**"; .EC replaces "**Figure**" by "**Equation**". Output is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the .af request of the formatter.

15. The user may wish to reset these registers after each first-level heading [4.6].

The *override* string is used to modify the normal numbering. If *flag* is omitted or 0, *override* is used as a prefix to the number; if *flag* is 1, *override* is used as a suffix; and if *flag* is 2, *override* replaces the number. For example, to produce figures numbered within sections, supply \n(H1 for *override* on each .FG call, and reset *Fg* at the beginning of each section, as shown in {4.6}.

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure and equation captions usually occur after the corresponding figures and equations.

7.6 Blocks of Filled Text

One can obtain blocks of filled text through the use of .DS or .DF. However, to have the block of filled text *centered* within the current line length, the *tbl*(I) program may be used:

```
   .
   .
   .
.DS 0 1
.TS
center;
lw40 .
T{
   .
   .
   .
T}
.TE
.DE
   .
   .
   .
```

The ".DS 0 1" begins a non-indented, filled display. The *tbl*(I) parameters set up a centered table with a column width of 40 ens. The "T{ ... T}" sequence allows filled text to be input as data within a table.

8. FOOTNOTES

There are two macros that delimit the text of footnotes,[16] a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters "*F" immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

8.2 Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
one or more lines of footnote text
.FE
```

The .FS (footnote start) marks the beginning of the text of the footnote, and the .FE marks its end. The *label* on the .FS, if present, will be used to mark the footnote text. Otherwise, the number retrieved from the string F will be used. Note that automatically-numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted *must* be followed by *label*, rather than by "*F". The text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are *not* permitted between the .FS and .FE macros. Examples:

16. Footnotes are processed in an environment that is different from that of the body of the text (see the .ev request in {9}).

1. Automatically-numbered footnote:

```
This is the line containing the word\*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

2. Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

The text of the footnote (enclosed within the .FS-.FE pair) should *immediately* follow the word to be footnoted in the input text, so that "*F" or *label* occurs at the end of a line of input and the next line is the .FS macro call. It is also good practice to append a unpaddable space [3.3] to "*F" or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Appendix C illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3 Format of Footnote Text •

.FD [arg] [1]

Within the footnote text, the user can control the formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left- or right-justification of the label when text indenting is used. The .FD macro is invoked to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is given by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the .ad, .hy, .na, and .nh requests in [9].

0	.nh	.ad	text indent	label left justified
1	.hy	.ad	"	"
2	.nh	.na	"	"
3	.hy	.na	"	"
4	.nh	.ad	no text indent	"
5	.hy	.ad	"	"
6	.nh	.na	"	"
7	.hy	.na	"	"
8	.nh	.ad	text indent	label right justified
9	.hy	.ad	"	"
10	.nh	.na	"	"
11	.hy	.na	"	"

If the first argument to .FD is out of range, the effect is as if .FD 0 were specified. If the first argument is omitted or null, the effect is equivalent to .FD 10 in *nroff* and to .FD 0 in *troff*; these are also the respective initial defaults.

If a second argument is specified, then whenever a first-level heading is encountered, automatically-numbered footnotes begin again with 1. This is most useful with the "section-page" page numbering scheme. As an example, the input line:

```
.FD "" 1
```

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading.

For long footnotes that continue onto the following page, it is possible that, if hyphenation is permitted, the last line of the footnote on the current page will be hyphenated. Except for this case (over which the user has control by specifying an *even* argument to .FD), hyphenation across pages is inhibited by PWB/MM.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In *troff*, footnotes are set in type that is two points smaller than the point size used in the body of the text.

8.4 Spacing between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register *Fs* to the desired value. For example:

```
.nr Fs 2
```

will cause two blank lines (a six-point vertical space) to occur between footnotes.

9. PAGE HEADERS AND FOOTERS

Text that occurs at the top of each page is known as the *page header*. Text printed at the bottom of each page is called the *page footer*. There can be up to three lines of text associated with the header: every page, even page only, and odd page only. Thus the page header may have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways of changing them. We use the term *header (not* qualified by *even* or *odd)* to mean the line of the page header that occurs on every page, and similarly for the term *footer*.

9.1 Default Headers and Footers

By default, each page has a centered page number as the header {9.2}. There is no default footer and no even/odd default headers or footers, except as specified in {9.9}.

In a memorandum or a released paper, the page header on the first page is automatically suppressed *provided* a break does *not* occur before .MT is called. The macros and text of {6.9} and of {9} as well as .nr and .ds requests do *not* cause a break and are permitted before the .MT macro call.

9.2 Page Header

```
.PH [arg]
```

For this and for the .EH, .OH, .PF, .EF, .OF macros, the argument is of the form:

```
"'left-part'center-part'right-part'"
```

If it is inconvenient to use the apostrophe (') as the delimiter (i.e., because it occurs within one of the parts), it may be replaced *uniformly* by *any* other character. On output, the parts are left-justified, centered, and right-justified, respectively. See {9.11} for examples.

The .PH macro specifies the header that is to appear at the top of every page. The initial value (as stated in {9.1}) is the default centered page number enclosed by hyphens. See the top of this page for an example of this default header.

If *debug mode* is set using the flag -rD1 on the command line {2.4}, additional information, printed at the top left of each page, is included in the default header. This consists of the SCCS [10] Release and Level of PWB/MM (thus identifying the current version {11.3}), followed by the current line number within the current input file.

9.3 Even-Page Header

```
.EH [arg]
```

The .EH macro supplies a line to be printed at the top of each even-numbered page, immediately *following* the header. The initial value is a blank line.

9.4 Odd-Page Header

.OH [arg]

This macro is the same as .EH, except that it applies to odd-numbered pages.

9.5 Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the -rC*n* flag is specified on the command line {2.4}, the type of copy *follows* the footer on a separate line. In particular, if -rC3 (DRAFT) is specified, then, in addition, the footer is initialized to contain the date {6.7.1}, instead of being a blank line.

9.6 Even-Page Footer

.EF [arg]

The .EF macro supplies a line to be printed at the bottom of each even-numbered page, immediately *preceding* the footer. The initial value is a blank line.

9.7 Odd-Page Footer

.OF [arg]

This macro is the same as .EF, except that it applies to odd-numbered pages.

9.8 Footer on the First Page

By default, the footer is a blank line. If, in the input text, one specifies .PF and/or .OF before the end of the first page of the document, then these lines will appear at the bottom of the first page.

The header (whatever its contents) *replaces* the footer *on the first page only* if the -rN1 flag is specified on the command line {2.4}.

9.9 Default Header and Footer with "Section-Page" Numbering

Pages can be numbered sequentially within sections {4.5}. To obtain this numbering style, specify -rN3 on the command line. In this case, the default *footer* is a centered "section-page" number, e.g. 3-5, and the default page header is blank.

9.10 Use of Strings and Registers in Header and Footer Macros •

String and register names may be placed in the arguments to the header and footer macros. If the value of the string or register is to be computed *when the respective header or footer is printed*, the invocation must be escaped by four (4) backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a .tl request during header or footer processing.

For example, the page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH "''Page \\\\nP'"
```

creates a right-justified header containing the word "Page" followed by the page number. Similarly, to specify a footer with the "section-page" style, one specifies (see {4.2.2.5} for meaning of *H1*):

```
.PF "''- \\\\n(H1-\\\\nP -'"
```

As another example, suppose that the user arranges for the string *a]* to contain the current section heading which is to be printed at the bottom of each page. The .PF macro call would then be:

```
.PF "''\\\\*(a]''"
```

If only one or two backslashes were used, the footer would print a constant value for *a]*, namely, its value when the .PF appeared in the input text.

9.11 Header and Footer Example •

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page:

```
.PH ""
.PF ""
.EH "'\\\\nP''Revision 3'"
.OH "'Revision 3''\\\\nP'"
```

9.12 Generalized Top-of-Page Processing •

☛ *This section is intended only for users accustomed to writing formatter macros.*

During header processing, PWB/MM invokes two user-definable macros. One, the .TP macro, is invoked in the environment (see .ev request in [9]) of the header; the other, .PX, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with "no-space" mode already in effect.

The effective initial definition of .TP (after the first page of a document) is:

```
.de TP
.sp
.tl \\*(}t
.if e 'tl \\*(}e
.if o 'tl \\*(}o
.sp
..
```

The string *}t* contains the header, the string *}e* contains the even-page header, and the string *}o* contains the odd-page header, as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause any desired header processing [11.5]. Note that formatting done within the .TP macro is processed in an environment different from that of the body.

For example, to obtain a page header that includes three centered lines of data, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

9.13 Generalized Bottom-of-Page Processing

The facility to permit user-defined processing for the bottom of each page is *not* currently available.

10. TABLE OF CONTENTS AND COVER SHEET

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively. The appropriate -rB*n* option [2.4] must *also* be specified on the command line. These macros should normally appear only once at the *end* of the document, after the Signature Block [6.11.1] and Notations [6.11.2] macros. They may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is therefore produced at the end.

10.1 Table of Contents

.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

The .TC macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the *Cl* register {4.4}. Note that -rB1 or -rB3 {2.4} must also be specified to the formatter on the command line. The arguments to .TC control the spacing before each entry, the placement of the associated page number, and additional text on the first page of the table of contents before the word "CONTENTS."

Spacing before each entry is controlled by the first two arguments; headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (½ a vertical space). Note that *slevel* does *not* control what levels of heading have been saved; the saving of headings is the function of the *Cl* register {4.4}.

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots ("leaders") separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to *tlevel* (default 2), the page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate the heading text from the page number. If *tab* is 0 (the default value), dots (i.e., leaders) are used; if *tab* is greater than 0, spaces are used. For headings whose level is greater than *tlevel*, the page numbers are separated from the heading text by two spaces (i.e., they are "ragged right").

All additional arguments (e.g., *head1*, *head2*, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the .TC macro is invoked with at most four arguments, then the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed. By defining .TX and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
..
.TC
```

yields:

Special Application
Message Transmission

Approved: _____

CONTENTS

:

10.2 Cover Sheet

.CS [pages] [other] [total] [figs] [tbls] [refs]

The .CS macro generates a cover sheet in either the TM or released-paper style.[17] All of the other information for the cover sheet is obtained from the data given before the .MT macro call {6.9}. If the released-paper style is used, all arguments to .CS are ignored. If a memorandum style is used, the .CS macro generates the "Cover Sheet for Technical Memorandum." The arguments provide the data that appears in the lower left corner of the TM cover sheet [2]: the number of pages of text, the number of other pages, the total number of pages, the number of figures, the number of tables, and the number of references.

11. MISCELLANEOUS FEATURES

11.1 Bold, Italic, and Roman

.B [bold-arg] [previous-font-arg]
.I [italic-arg] [previous-font-arg]
.R

When called without arguments, .B (or .I) changes the font to bold (or italic) in *troff*, and initiates underlining in *nroff*.[18] This condition continues until the occurrence of a .R, when the regular roman font is restored. Thus,

```
.I
here is some text.
.R
```

yields:

here is some text.

If .B or .I is called with one argument, that argument is printed in the appropriate font (underlined in *nroff*). Then the *previous* font is restored (underlining is turned off in *nroff*). If two arguments are given to a .B or .I, the second argument is then concatenated to the first with no intervening space, but is printed in the previous font (not underlined in *nroff*). For example:

```
.I italic
text
.I right -justified
```

produces:

italic text *right*-justified

One can use both bold and italic fonts if one intends to use *troff*, but the *nroff* version of the output does not distinguish between bold and italic. It is probably a good idea to use .I only, unless bold is truly required. Note that font changes in headings are handled separately {4.2.2.4.1}.

Anyone using a terminal that cannot underline might wish to insert:

```
.rm ul
.rm cu
```

at the beginning of the document to eliminate *all* underlining.

11.2 Justification of Right Margin

.SA [arg]

The .SA macro is used to set right-margin justification for the main body of text. Two justification flags are used: *current* and *default*. .SA 0 sets both flags to no justification, i.e., it acts like the .na request. .SA 1 is the inverse: it sets both flags to cause justification, just like the .ad request. However, calling

17. But only if -rB2 or -rB3 has been specified on the command line.

18. For ease of explanation, in this section {11.1} *troff* behavior is described first, the convention of {1.2} not withstanding.

.SA *without* an argument causes the *current* flag to be copied from the *default* flag, thus performing either a .na or .ad, depending on what the *default* is. Initially, both flags are set for no justification in *nroff* and for justification in *troff*.

In general, the request .na can be used to ensure that justification is turned off, but .SA should be used to restore justification, rather than the .ad request. In this way, justification or lack thereof for the remainder of the text is specified by inserting .SA 0 or .SA 1 *once* at the beginning of the document.

11.3 SCCS Release Identification

The string *RE* contains the SCCS [10] Release and Level of the current version of PWB/MM. For example, typing:

```
This is version \*(RE of the macros.
```

produces:

This is version 12.2 of the macros.

This information is useful in analyzing suspected bugs in PWB/MM. The easiest way to have this number appear in your output is to specify -rD1 {2.4} on the command line, which causes the string *RE* to be output as part of the page header {9.2}.

11.4 Two-Column Output

PWB/MM can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The .2C macro begins two-column processing which continues until a .1C macro is encountered. In two-column processing, each physical page is thought of as containing two columnar "pages" of equal (but smaller) "page" width. Page headers and footers are *not* affected by two-column processing. The .1C macro does *not* "balance" two-column output.

11.5 Column Headings for Two-Column Output •

☛ *This section is intended only for users accustomed to writing formatter macros.*

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page {9}. This is accomplished by redefining the .TP macro {9.12} to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\nP'OVERALL"
.tl "TITLE"
.sp
.nf
.ta 16C 31R 34 50C 65R
left—center—right—left—center—right                    (where — stands for the tab character)
—first column———second column
.fi
.sp 2
..
```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

11.6 Vertical Spacing

```
.SP [lines]
```

There exist several ways of obtaining vertical spacing, all with different effects.

The .sp request spaces the number of lines specified, *unless* "no space" (.ns) mode is on, in which case the request is ignored. This mode is typically set at the end of a page header in order to eliminate spacing by a .sp or .bp request that just happens to occur at the top of a page. This mode can be turned *off* via the .rs ("restore spacing") request.

The .SP macro is used to avoid the accumulation of vertical space by successive macro calls. Several .SP calls in a row produce *not* the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```
.SP 2
.SP 3
.SP
```

Many PWB/MM macros utilize .SP for spacing. For example, ".LE 1" {5.3.2} immediately followed by ".P" {4.1} produces only a single blank line (½ a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (*one* vertical space). Unscaled fractional amounts are permitted; like .sp, .SP is also inhibited by the .ns request.

11.7 Skipping Pages

.SK [pages]

The .SK macro skips pages, but retains the usual header and footer processing. If *pages* is omitted, null, or 0, .SK skips to the top of the next page *unless* it is currently at the top of a page, in which case it does nothing. .SK *n* skips *n* pages. That is, .SK always positions the text that follows it at the top of a page, while .SK 1 always leaves one page that is blank except for the header and footer.

11.8 Setting Point Size and Vertical Spacing

In *troff*, the default point size (obtained from the register *S* {2.4}) is 10, with a vertical spacing of 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing may be changed by invoking the .S macro:

.S [arg]

If *arg* is null, the *previous* point size is restored. If *arg* is negative, the point size is decremented by the specified amount. If *arg* is *signed* positive, the point size is incremented by the specified amount, and if *arg* is unsigned, it is used as the new point size; if *arg* is greater than 99, the *default* point size (10) is restored. Vertical spacing is always two points greater than the point size.[19]

12. ERRORS AND DEBUGGING

12.1 Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.
- To avoid confusion regarding the location of the error, the formatter output buffer (which may contain some text) is printed.
- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line number (in the current input file) of the last processed input line. (All the error messages are explained in Appendix E.)
- Processing terminates, unless the register *D* {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

☛ *The error message is printed by writing it directly to the user's terminal. If an output filter, such as* gsi(1), 450(1), *or* hp(1) *is being used to post-process* nroff *output, the message may be garbled by being intermixed with text held in that filter's output buffer.*

19. Footnotes {8} are printed in a size two points *smaller* than the point size of the body, with an additional vertical spacing of three points between footnotes.

☛ *If either* tbl(I) *or* eqn(I)/neqn(I), *or both are being used, and if the* -olist *option of the formatter causes the last page of the document* not *to be printed, a harmless "broken pipe" message results.*

12.2 Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing .FE or .DE). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing .DE or .FE, the appropriate action is to search backwards from the termination point looking for the corresponding .DS, .DF, or .FS.

The following command:

grep -n "^\.[EDFT][EFNQS]" files ...

prints all the .DS, .DF, .DE, .FS, .FE, .TS, .TE, .EQ, and .EN macros found in *files* ..., each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

13. EXTENDING AND MODIFYING THE MACROS •

13.1 Naming Conventions

In this section, the following conventions are used to describe legal names:

n: digit
a: lower-case letter
A: upper-case letter
x: any letter or digit (any alphanumeric character)
s: special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that *request, macro,* and *string* names are kept by the formatters in a single internal table, so that there must be no duplication among such names. *Number register* names are kept in a separate table.

13.1.1 Names Used by Formatters.

requests: aa (most common)
an (only one, currently: .c2)

registers: aa (normal)
.x (normal)
.s (only one, currently: .S)
% (page number)

13.1.2 Names Used by PWB/MM.

macros: AA (most common, accessible to user)
A (less common, accessible to user)
)x (internal, constant)
>x (internal, dynamic)

strings: AA (most common, accessible to user)
A (less common, accessible to user)
]x (internal, usually allocated to specific functions throughout)
}x (internal, more dynamic usage)

registers: Aa (most common, accessible to users)
An (common, accessible to user)
A (accessible, set on command line)
:x (mostly internal, rarely accessible, usually dedicated)
;x (internal, dynamic, temporaries)

13.1.3 Names Used by EQN/NEQN and TBL. The equation preprocessors, *eqn*(I) and *neqn*(I), use registers *and* string names of the form *nn*. The table preprocessor, *tbl*(I), uses names of the form:

a- a+ a| nn #a ## #- #^ ^a T& TW

13.1.4 User-Definable Names. After the above, what is left for user extensions? To avoid problems, we suggest using names that consist either of a single lower-case letter, or of a lower-case letter followed by anything other than a lower-case letter. The following is a sample naming convention:

macros:	aA
	Aa
strings:	a
	a) (or a]; or a}, etc.)
registers	a
	aA

13.2 Sample Extensions

13.2.1 Appendix Headings. The following gives a way of generating and numbering appendices:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH "'''Appendix \\na - \\\\\\\\nP'"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form ".aH "title"" begins a new page (with the page header changed to "Appendix *a* - *n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish Appendix titles to be centered must, in addition, set the register *Hc* to 1 {4.2.2.3}.

13.2.2 Hanging Indent with Tabs. The following example illustrates the use of the hanging-indent feature of variable-item lists {5.3.3.6}. First, a user-defined macro is built to accept four arguments that make up the *mark*. Each argument is to be separated from the previous one by a tab character; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the "\&" is used so that the formatter will not interpret such a line as a formatter request or macro.[20] The "\t" is translated by the formatter into a tab character. The "\c" is used to concatenate the line of *text* that follows the macro to the line of text built by the macro. The macro definition and an example of its use are as follows:

20. The two-character sequence "\&" is understood by the formatters to be a "zero-width" space, i.e., it causes no output characters to appear.

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
..
  :
  :
.ta 9n 18n 27n 36n
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\□c none none no                    (□ stands for a space)
Hyphenation indicator character is set to "c" or removed.
During text processing the indicator is suppressed
and will not appear in the output.
Prepending the indicator to a word has the effect
of preventing hyphenation of that word.
.LE
```

The resulting output is:

.nh	off	–	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	–	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to "c" or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

14. CONCLUSION

The following are the qualities that we have tried to emphasize in PWB/MM, in approximate order of importance:

- *Robustness in the face of error*—A user need not be an *nroff/troff* expert to use these macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error, or a message describing the error is produced. We have tried to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce simple documents. Reasonable default values are provided, where at all possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt the output to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—We have made a strong effort to use mnemonic naming conventions and consistent techniques in the construction of the macros. Naming conventions are given so that a user can add new macros or redefine existing ones, if necessary.
- *Device independence*—The most common use of PWB/MM is to print documents on hard-copy typewriter terminals, using the *nroff* formatter. The macros can be used conveniently with both 10- and

12-pitch terminals. In addition, output can be scanned with an appropriate CRT terminal. The macros have been constructed to allow compatibility with *troff*, so that output can be produced both on typewriter-like terminals and on a phototypesetter.

- *Minimization of input*—The design of the macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, he or she need only set a specific parameter *once* at the beginning of a document, rather than add a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text, although the user may obtain a number of output styles by setting a few global flags. For example, the .H macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or, for that matter, within a single document.

Future releases of PWB/MM will provide additional features that are found to be useful. The authors welcome comments, suggestions, and criticisms of the macros and of this manual.

Acknowledgements. We are indebted to T. A. Dolotta for his continuing guidance during the development of PWB/MM. We also thank our many users who have provided much valuable feedback, both about the macros and about this manual. Many of the features of PWB/MM are patterned after similar features in a number of earlier macro packages, and, in particular, after one implemented by M. E. Lesk. Finally, because PWB/MM often approaches the limits of what is possible with the text formatters, during the implementation of PWB/MM we have generated atypical requirements and encountered unusual problems; we thank J. F. Ossanna for his willingness to add new features to the formatters and to invent ways of having the formatters perform unusual but desired actions.

References

[1] Dolotta, T. A., Haight, R. C., and Piskorik, E. M., eds. *PWB/UNIX User's Manual—Edition 1.0.* Bell Laboratories, May 1977.

[2] Bell Laboratories, Methods and Systems Department. Office Guide. Unpublished Memorandum, Bell Laboratories, April 1972 (as revised).

[3] Kernighan, B. W. UNIX for Beginners. Bell Laboratories, October 1974.

[4] Kernighan, B. W. A Tutorial Introduction to the UNIX Text Editor. Bell Laboratories, October 1974.

[5] Kernighan, B. W. A TROFF Tutorial. Bell Laboratories, August 1976.

[6] Kernighan, B. W., and Cherry, L. L. Typesetting Mathematics—User's Guide (Second Edition). Bell Laboratories, June 1976.

[7] Scrocca, C. New Graphic Symbols for EQN and NEQN. Bell Laboratories, September 1976.

[8] Smith, D. W., and Piskorik, E. M. Typing Documents with PWB/MM. Bell Laboratories, October 1977.

[9] Ossanna, J. F. NROFF/TROFF User's Manual. Bell Laboratories, October 1976.

[10] Bonanni, L. E., and Glasser, A. L. SCCS/PWB User's Manual. Bell Laboratories, November 1977.

[11] Lesk, M. Tbl—A Program to Format Tables. Bell Laboratories, September 1977.

Appendix A: DEFINITIONS OF LIST MACROS •

☛ *This appendix is intended only for users accustomed to writing formatter macros.*

Here are the definitions of the list-initialization macros {5.3.3}:[21]

```
.de AL
.if!@\\$1@@ .if!@\\$1@1@ .if!@\\$1@a@ .if!@\\$1@A@ .if!@\\$1@I@ .if!@\\$1@i@ .)D"AL:bad arg:\\$1
.if \\n(.$<3 \{.ie \w@\\$2@=0 .)L \\n(Lin 0 \\n(Lin-\w@\0\0.@u 1 "\\$1"
.el .LB 0\\$2 0 2 1 "\\$1" \}
.if \\n(.$>2 \{.ie \w@\\$2@=0 .)L \\n(Lin 0 \\n(Lin-\w@\0\0.@u 1 "\\$1" 0 1
.el .LB 0\\$2 0 2 1 "\\$1" 0 1 \}
..
.de BL
.nr ;0 \\n(Pi
.if \\n(.$>0 .if \w@\\$1@>0 .nr ;0 0\\$1
.if \\n(.$<2 .LB \\n(;0 0 1 0 \\•(BU
.if \\n(.$>1 .LB \\n(;0 0 1 0 \\•(BU 0 1
.rr ;0
..
.de DL
.nr ;0 \\n(Pi
.if \\n(.$>0 .if \w@\\$1@>0 .nr ;0 0\\$1
.if \\n(.$<2 .LB \\n(;0 0 1 0 \(em
.if \\n(.$>1 .LB \\n(;0 0 1 0 \(em 0 1
.rr ;0
..
.de ML
.if !\\n(.$ .)D "ML:missing arg"
.nr ;0 \w@\\$1@u/3u/\\n(.su+1u\" get size in n's
.if !\\n(.$-1 .LB \\n(;0 0 1 0 "\\$1"
.if \\n(.$-1 .if !\\n(.$-2 .LB 0\\$2 0 1 0 "\\$1"
.if \\n(.$-2 .if !\w@\\$2@ .LB \\n(;0 0 1 0 "\\$1" 0 1
.if \\n(.$-2 .if \w@\\$2@ .LB 0\\$2 0 1 0 "\\$1" 0 1
..
.de RL
.nr ;0 6
.if \\n(.$>0 .if \w@\\$1@>0 .nr ;0 0\\$1
.if \\n(.$<2 .LB \\n(;0 0 2 4
.if \\n(.$>1 .LB \\n(;0 0 2 4 1 0 1
.rr ;0
..
.de VL
.if !\\n(.$ .)D "VL:missing arg"
.if !\\n(.$-2 .LB 0\\$1 0\\$2 0 0
.if \\n(.$-2 .LB 0\\$1 0\\$2 0 0 \& 0 1
..
```

Any of these can be redefined to produce different behavior: e.g., to provide two spaces between the bullet of a bullet item and its text, redefine .BL as follows before invoking it:[22]

```
.de BL
.LB 3 0 2 0 \\•(BU
..
```

---

21. On this page, @ represents the BEL character, .)D is an internal PWB/MM macro that prints error messages, and .)L is similar to .LB, except that it expects its arguments to be scaled.

22. With this redefinition, .BL cannot have any arguments.

Appendix B: USER-DEFINED LIST STRUCTURES •

☛ *This appendix is intended only for users accustomed to writing formatter macros.*

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level list nesting in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to have the following appearance:

A.
[1]
•
a)
+

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, you should know that the number register :g is used by the PWB/MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments :g, and each .LE call decrements it.

```
.de aL
'\"        register g is used as a local temporary to save :g before it is changed below
.nr g \\n(:g
.if \\ng=0 .AL A \" give me an A.
.if \\ng=1 .LB \\n(Li 0 1 4 \" give me a [1]
.if \\ng=2 .BL \" give me a bullet
.if \\ng=3 .LB \\n(Li 0 2 2 a \" give me an a)
.if \\ng=4 .ML + \" give me a +
..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
first line.
.aL
.LI
second line.
.LE
.LI
third line.
.LE
```

will yield:

A. first line.

[1] second line.

B. third line.

There is another approach to lists that is similar to the .H mechanism. The list-initialization, as well as the .LI and the .LE macros are all included in a single macro. That macro (called .bL below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item:

```
.de bL
.ie \\n(.$ .nr g \\$1 \" if there is an argument, that is the level
.el .nr g \\n(:g \" if no argument, use current level
.if \\ng-\\n(:g>1 .)D "**ILLEGAL SKIPPING OF LEVEL" \" increasing level by more than 1
.if \\ng>\\n(:g \{.aL \\ng-1 \" if g > :g, begin new list
.       nr g \\n(:g\} \" and reset g to current level (.aL changes g)
.if \\n(:g>\\ng .LC \\ng \" if :g > g, prune back to correct level
'\"       if :g = g, stay within current list
.LI \" in all cases, get out an item
..
```

For .bL to work, the previous definition of the .aL macro must be changed to obtain the value of *g* from its argument, rather than from *:g*. Invoking .bL without arguments causes it to stay at the current list level. The PWB/MM .LC macro (List Clear) removes list descriptions until the level is less than or equal to that of its argument. For example, the .H macro includes the call ".LC 0". If text is to be resumed at the end of a list, insert the call ".LC 0" to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC 0
fifth line.
```

yields:

The quick brown fox jumped over the lazy dog's back.

A. first line.

   [1] second line.

B. third line.

C. fourth line.

fifth line.

Appendix C: SAMPLE FOOTNOTES

The following example illustrates several footnote styles and both labeled and automatically-numbered footnotes. The actual input for the immediately following text and for the footnotes at the bottom of this page is shown on the following page:

With the footnote style set to the *nroff* default, we process a footnote[1] followed by another one.[*****] Using the .FD macro, we changed the footnote style to hyphenate, right margin justification, indent, and left justify the label. Here is a footnote,[2] and another.[†] The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification, no indentation, and with the label left-justified. Here comes the final one.[3]

---

1. This is the first footnote text example (.FD 10). This is the default style for *nroff*. The right margin is *not* justified. Hyphenation is *not* permitted. The text is indented, and the automatically generated label is *right*-justified in the text-indent space.

***** This is the second footnote text example (.FD 10). This is also the default *nroff* style but with a long footnote label provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, the label is *left*-justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted. The quick brown fox jumped over the lazy dog's back.

† This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is *not* justified, hyphenation is *not* permitted, the footnote text is *not* indented, and the label is placed at the beginning of the first line. The quick brown fox jumped over the lazy dog's back. Now is the time for all good men to come to the aid of their country.

```
.FD 10
With the footnote style set to the
.I nroff
default, we process a footnote\*F
.FS
This is the first footnote text example (.FD 10).
This is the default style for
.I nroff.
The right margin is
.I not
justified.
Hyphenation is
.I not
permitted.
The text is indented, and the automatically generated label is
.I right -justified
in the text-indent space.
.FE
followed by another one.*****\□                                    (□ stands for a space)
.FS *****
This is the second footnote text example (.FD 10).
This is also the default
.I nroff
style but with a long footnote label provided by the user.
.FE
.FD 1
Using the .FD macro, we changed the footnote style to hyphenate, right margin justification,
indent, and left justify the label.
Here is a footnote,\*F
.FS
This is the third footnote example (.FD 1).
The right margin is justified, the footnote text is indented, the label is
.I left -justified
in the text-indent space.
Although not necessarily illustrated by this example, hyphenation is permitted.
The quick brown fox jumped over the lazy dog's back.
.FE
and another.\(dg\□
.FS \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification,
no indentation, and with the label left-justified.
Here comes the final one.\*F\□
.FS
This is the fifth footnote example (.FD 6).
The right margin is
.I not
justified, hyphenation is
.I not
permitted, the footnote text is
.I not
indented, and the label is placed at the beginning of the first line.
The quick brown fox jumped over the lazy dog's back.
Now is the time for all good men to come to the aid of their country.
.FE
```

Appendix D: SAMPLE LETTER

☛ *The* nroff *and* troff *outputs corresponding to the input text below are shown on the following pages.*

```
.ND "November 1, 1977"
.TL 334455
Out-of-Hours Course Description
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
.MT 0
.DS
J. M. Jones:
.DE
.P
Please use the following description for the Out-of-Hours course
"Document Preparation on the PWB/UNIX•
.FS •
UNIX is a Trademark of Bell Laboratories.
.FE
time-sharing system":
.P
The course is intended for clerks, typists, and others
who intend to use the PWB/UNIX system
for preparing documentation.
The course will cover such topics as:
.VL 18
.LI Environment:
utilizing a time-sharing computer system;
accessing the system;
using appropriate output terminals.
.LI Files:
how text is stored on the system;
directories;
manipulating files.
.LI "Text editing:"
how to enter text so that subsequent revisions are easier to make;
how to use the editing system to
add, delete, and move lines of text;
how to make corrections.
.LI "Text processing:"
basic concepts;
use of general-purpose formatting packages.
.LI "Other facilities:"
additional capabilities useful to the typist such as the
.I "typo, spell, diff,"
and
.I grep
commands and a desk-calculator package.
.LE
.SG jrm
.NS
S. P. Lename
H. O. Del
M. Hill
.NE
```

Bell Laboratories

subject: Out-of-Hours Course Description
Case: 334455

date: November 1, 1977

from: D. W. Stevenson
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course "Document Preparation on the PWB/UNIX* time-sharing system":

The course is intended for clerks, typists, and others who intend to use the PWB/UNIX system for preparing documentation. The course will cover such topics as:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the typo, spell, diff, and grep commands and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a Trademark of Bell Laboratories.

Bell Laboratories

subject: **Out-of-Hours Course Description**
Case: **334455**

date: **November 1, 1977**

from: **D. W. Stevenson**
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course "Document Preparation on the PWB/UNIX* time-sharing system":

The course is intended for clerks, typists, and others who intend to use the PWB/UNIX system for preparing documentation. The course will cover such topics as:

Environment:	utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
Files:	how text is stored on the system; directories; manipulating files.
Text editing:	how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
Text processing:	basic concepts; use of general-purpose formatting packages.
Other facilities:	additional capabilities useful to the typist such as the *typo*, *spell*, *diff*, and *grep* commands and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a Trademark of Bell Laboratories.

Appendix E: ERROR MESSAGES

I. PWB/MM Error Messages

Each PWB/MM error message consists of a standard part followed by a variable part. The standard part is of the form:

ERROR:input line *n*:

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation:[23]

Check TL, AU, AS, AE, MT sequence	The proper sequence of macros for the beginning of a memorandum is shown in {6.9}. Something has disturbed this order.
AL:bad arg:value	The argument to the .AL macro is not one of 1, A, a, I, or i. The incorrect argument is shown as *value*.
CS:cover sheet too long	The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.4}.
DS:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.
DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
DS:missing DE	.DS or .DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
FE:no FS	.FE has been encountered with no previous .FS to match it.
FS:illegal inside TL or AS	.FS-.FE pair cannot be used inside the memorandum title or abstract.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
H:bad arg:value	The first argument to .H must be a single digit from 1 to 7, but *value* has been supplied instead.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing arg	.H needs at least 1 argument.
HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro {5.3.3}. Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.

23. This list is set up by ".LB 37 0 2 0" {5.4}.

LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as *value*.
SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.

II. Formatter Error Messages

Most messages issued by the formatter are self-explanatory. Those error messages over which the *user* has (some) control are listed below. Any other error messages should be reported to the local system-support group.

"Cannot open *filename*" is issued if one of the files in the list of files to be processed cannot be opened. If the filename is of the form /usr/lib/tmac.*name*, then the option -m*name* specifies an incorrect *name*. If the filename is of the form /usr/lib/term/*name*, then the *nroff* option -T*name* is incorrect. If the filename is of the form /usr/lib/font/*xx*, then the font specified in a formatter .fp request is incorrect.

"Exception word list full" indicates that too many words have been specified in the hyphenation exception list (via .hw requests).

"Line overflow" means that the output line being generated was too long for the formatter's line buffer. The excess was discarded. See the "Word overflow" message below.

"Out of temp file space" means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing ".."), or a huge table of contents.

"Too many page numbers" is issued when the list of pages specified to the formatter *-o* option is too long.

"Too many string/macro names" is issued when the pool of string and macro names is full. Unneeded strings and macros can be deleted using the .rm request.

"Too many number registers" means that the pool of number register names is full. Unneeded registers can be deleted by using the .rr request.

"Word overflow" means that a word being generated exceeded the formatter's word buffer. The excess characters were discarded. A likely cause for this and for the "Line overflow" message above are very long lines or words generated through the misuse of \c or of the .cu request, or very long equations produced by *eqn*(1)/*neqn*(1).

Appendix F: SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS

I. Macros

The following is an alphabetical list of macro names used by PWB/MM. The first line of each item gives the name of the macro, a brief description, and a reference to the section in which the macro is described. The second line gives a prototype call of the macro.

Macros marked with an asterisk are *not*, in general, invoked directly by the user. Rather, they are "user exits" called from inside header, footer, or other macros.

1C — One-column processing {11.4}
.1C

2C — Two-column processing {11.4}
.2C

AE — Abstract end {6.4}
.AE

AF — Alternate format of "Subject/Date/From" block {6.7.2}
.AF [company-name]

AL — Automatically-incremented list start {5.3.3.1}
.AL [type] [text-indent] [1]

AS — Abstract start {6.4}
.AS [arg] [indent]

AU — Author information {6.2}
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]

B — Bold (underline in *nroff*) {11.1}
.B [bold-arg] [previous-font-arg]

BL — Bullet list start {5.3.3.2}
.BL [text-indent] [1]

CS — Cover sheet {10.2}
.CS [pages] [other] [total] [figs] [tbls] [refs]

DE — Display end {7.1}
.DE

DF — Display floating start {7.2}
.DF [format] [fill]

DL — Dash list start {5.3.3.3}
.DL [text-indent] [1]

DS — Display static start {7.1}
.DS [format] [fill]

EC — Equation caption {7.5}
.EC [title] [override] [flag]

EF — Even-page footer {9.6}
.EF [arg]

EH — Even-page header {9.3}
.EH [arg]

EN — End equation display {7.4}
.EN

EQ — Equation display start {7.4}
.EQ

FD Footnote default format {8.3}
.FD [arg] [1]

FE Footnote end {8.2}
.FE

FG Figure title {7.5}
.FG [title] [override] [flag]

FS Footnote start {8.2}
.FS [label]

H Heading—numbered {4.2}
.H level [heading-text]

HC Hyphenation character {3.4}
.HC [hyphenation-indicator]

HM Heading mark style (Arabic or Roman numerals, or letters) {4.2.2.5}
.HM [arg1] ... [arg7]

HU Heading—unnumbered {4.3}
.HU heading-text

HX * Heading user exit X (before printing heading) {4.6}
.HX dlevel rlevel heading-text

HZ * Heading user exit Z (after printing heading) {4.6}
.HZ dlevel rlevel heading-text

I Italic (underline in *nroff*) {11.1}
.I [italic-arg] [previous-font-arg]

LB List begin {5.4}
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]

LC List-status clear {Appendix B}
.LC [list-level]

LE List end {5.3.2}
.LE [1]

LI List item {5.3.1}
.LI [mark] [1]

ML Marked list start {5.3.3.4}
.ML mark [text-indent] [1]

MT Memorandum type {6.6}
.MT [type] [1]

ND New date {6.7.1}
.ND new-date

NE Notation end {6.11.2}
.NE

NS Notation start {6.11.2}
.NS [arg]

OF Odd-page footer {9.7}
.OF [arg]

OH Odd-page header {9.4}
.OH [arg]

OK Other keywords for TM cover sheet {6.5}
.OK [keyword] ...

P — Paragraph {4.1}
.P [type]

PF — Page footer {9.5}
.PF [arg]

PH — Page header {9.2}
.PH [arg]

PX * — Page-header user exit {9.12}
.PX

R — Return to regular (roman) font (end underlining in *nroff*) {11.1}
.R

RL — Reference list start {5.3.3.5}
.RL [text-indent] [1]

S — Set *troff* point size and vertical spacing {11.8}
.S [arg]

SA — Set adjustment (right-margin justification) default {11.2}
.SA [arg]

SG — Signature line {6.11.1}
.SG [arg] [1]

SK — Skip pages {11.7}
.SK [pages]

SP — Space—vertically {11.6}
.SP [lines]

TB — Table title {7.5}
.TB [title] [override] [flag]

TC — Table of contents {10.1}
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

TE — Table end {7.3}
.TE

TL — Title of memorandum {6.1}
.TL [charging-case] [filing-case]

TM — Technical Memorandum number(s) {6.3}
.TM [number] ...

TP * — Top-of-page macro {9.12}
.TP

TS — Table start {7.3}
.TS

TX * — Table-of-contents user exit {10.1}
.TX

VL — Variable-item list start {5.3.3.6}
.VL text-indent [mark-indent] [1]

II. Strings

The following is an alphabetical list of string names used by PWB/MM, giving for each a brief description, section reference, and initial (default) value(s). See {1.4} for notes on setting and referencing strings.

BU Bullet {3.7}
nroff: ⊖
troff: •

F Footnote numberer {8.1}
nroff: \u\\n+ (:p\d
troff: \v'-.4m'\s-3\\n+ (:p\s0\v'.4m'

DT Date (current date, unless overridden) {6.7.1}
Month day, year (e.g., October 31, 1977)

HF Heading font list, up to seven codes for heading levels 1 through 7 {4.2.2.4.1}
3 3 2 2 2 2 2 (all underlined in *nroff*, and B B I I I I I in *troff*)

RE SCCS Release and Level of PWB/MM {11.3}
Release.Level (e.g., 12.2)

Note that if the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.8}.

III. Number Registers

This section provides an alphabetical list of register names, giving for each a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive).

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set *only* from the command line or *before* the PWB/MM macro definitions are read by the formatter {2.4, 2.5}. See {1.4} for notes on setting and referencing registers.

A * Has the effect of invoking the .AF macro without an argument {2.4}
0, [0:1]

Au Inhibits printing of author's location, department, room, and extension in the "from" portion of a memorandum {6.2}
1, [0:1]

B * Defines table-of-contents and/or cover-sheet macros {2.4}
0, [0:3]

C * Copy type (Original, DRAFT, etc.) {2.4}
0 (Original), [0:3]

Cl Contents level (i.e., level of headings saved for table of contents) {4.4}
2, [0:7]

D * Debug flag {2.4}
0, [0:1]

Ds Static display pre- and post-space {7.1}
1, [0:1]

Ec Equation counter, used by .EC macro {7.5}
0, [0:?], incremented by 1 for each .EC call.

Ej Page-ejection flag for headings {4.2.2.1}
0 (no eject), [0:7]

Fg Figure counter, used by .FG macro {7.5}
0, [0:?], incremented by 1 for each .FG call.

Fs Footnote space (i.e., spacing between footnotes) {8.4}
1, [0:?]

H1-H7 Heading counters for levels 1-7 {4.2.2.5}
0, [0:?], incremented by .H of corresponding level or .HU if at level given by register *Hu*. H2-H7 are reset to 0 by any heading at a lower-numbered level.

Hb Heading break level (after .H and .HU) {4.2.2.2}
2, [0:7]

Hc Heading centering level for .H and .HU {4.2.2.3}
0 (no centered headings), [0:7]

Hi Heading temporary indent (after .H and .HU) {4.2.2.2}
1 (indent as paragraph), [0:2]

Hs Heading space level (after .H and .HU) {4.2.2.2}
2 (space only after .H 1 and .H 2), [0:7]

Ht Heading type (for .H: single or concatenated numbers) {4.2.2.5}
0 (concatenated numbers: 1.1.1, etc.), [0:1]

Hu Heading level for unnumbered heading (.HU) {4.3}
2 (.HU at the same level as .H 2), [0:7]

Hy Hyphenation control for body of document {3.4}
1 (automatic hyphenation on), [0:1]

L * Length of page {2.4}
66, [20:?] (11i, [2i:?] in *troff*)[24]

Li List indent {5.3.3.1}
5, [0:?]

N * Numbering style {2.4}
0, [0:3]

O * Offset of page {2.4}
0, [0:?] (0.5i, [0i:?] in *troff*)[24]

P Page number, managed by PWB/MM {2.4}
0, [0:?]

Pi Paragraph indent {4.1}
5, [0:?]

Pt Paragraph type {4.1}
2 (paragraphs indented except after headings, lists, and displays), [0:2]

S * *Troff* default point size {2.4}
10, [6:36]

Si Standard indent for displays {7.1}
5, [0:?]

T * Type of *nroff* output device {2.4}
0, [0:2]

Tb Table counter {7.5}
0, [0:?], incremented by 1 for each .TB call.

U * Underlining style (*nroff*) for .H and .HU {2.4}
0 (continuous underline when possible), [0:1]

W * Width of page (line and title length) {2.4}
65, [10:1365] (6.5i, [2i:7.54i] in *troff*)[24]

---

24 For *nroff*, these values are *unscaled* numbers representing lines or character positions; for *troff*, these values must be *scaled*.

Typing Documents with PWB/MM

D. W. Smith and E. M. Piskorik

Bell Laboratories
Piscataway, New Jersey 08854

This guide shows several examples of documents prepared with PWB/MM, a set of general-purpose formatting macros used with the PWB/UNIX* text formatters *nroff* and *troff* (as well as with the *eqn/neqn* and *tbl* programs) to produce memoranda, letters, books, manuals, etc. References to manuals for these programs are given on p. 16.

In the examples, input is shown in this
Helvetica sans serif font.

The resulting output is shown (boxed) in this
Times Roman font.

Substitutable arguments are shown in this
Times Roman Italic font.

Square brackets (*[...]*) indicate that the enclosed substitutable argument is optional.

All output shown in the examples was done by *troff*; *nroff* output would look somewhat different.†

Contents

* UNIX is a Trademark of Bell Laboratories.

† For example, what we call a "blank line" is a blank line in *nroff*, but is ½ of a vertical space in *troff*, while headings that are underlined in *nroff* are either bold or *italic* in *troff*.

Paragraphs and Headings

☛ *The output for the following is shown on p. 3.*

```
.H 1 "PARAGRAPHS AND HEADINGS"
This section describes the types of paragraphs
and the kinds of headings that are available.
.H 2 Paragraphs
Paragraphs are specified by the .P macro.
Usually, they are indented except
after headings, lists, and displays.
The number register Pt is used
to change the paragraph style.
.H 2 Headings
.H 3 "Numbered Headings."
There are seven levels of numbered headings.
Level 1 is the most major or highest;
level 7, the lowest.
.P
Headings are specified with the .H macro, whose
first argument is the level of heading (1 through 7).
.P
The appearance of headings varies according to
the level.
On output, level 1-headings are preceded by two
blank lines; all others are preceded by one blank line.
Level 1 and level 2 headings produce stand-alone
headings, underlined in
.I nroff
and bold in
.I troff.
Levels 3 through 7 are run-in and underlined (or Italic).
.H 3 "Unnumbered Headings."
The macro .HU is a special case
of .H, in that no heading number is printed.
Each .HU heading has the level given by
the register Hu, whose initial value is 2.
Usually, the value of that register is
set to make unnumbered headings (if any) occur
at the lowest heading level in a document.
```

Paragraph and Heading Parameters

There are many parameters that can change the output appearance of headings and paragraphs. Given below are some of these parameters, their *default* values, and their meanings (level 1 is the *most major* or *highest*, while level 7 is the *lowest*):

.nr Pi 5 paragraph-indent in characters (or ens).
.nr Pt 0 never indent paragraphs.
.nr Pt 1 always indent paragraphs.
.nr Pt 2 indent paragraphs *except* after headings, lists, and displays (*default*).
.ds HF 3 3 2 2 2 2 2
font specification for each of the 7 heading levels:
1 indicates roman,
2 indicates italic,
3 indicates bold.

-1-

1. PARAGRAPHS AND HEADINGS

This section describes the types of paragraphs and the kinds of headings that are available.

1.1 Paragraphs

Paragraphs are specified by the .P macro. Usually, they are indented except after headings, lists, and displays. The number register Pt is used to change the paragraph style.

1.2 Headings

1.2.1 Numbered Headings. There are seven levels of numbered headings. Level 1 is the most major or highest; level 7, the lowest.

Headings are specified with the .H macro, whose first argument is the level of heading (1 through 7).

The appearance of headings varies according to the level. On output, level 1 headings are preceded by two blank lines; all others are preceded by one blank line. Level 1 and level 2 headings produce stand-alone headings, underlined in *nroff* and bold in *troff*. Levels 3 through 7 are run-in and underlined (or italic).

1.2.2 Unnumbered Headings. The macro .HU is a special case of .H, in that no heading number is printed. Each .HU heading has the level given by the register Hu, whose initial value is 2. Usually, the value of that register is set to make unnumbered headings (if any) occur at the lowest heading level in a document.

.HM 1 1 1 1 1 1 1
"marking" style for each heading level; the above yields an all-numeric marking style. Available styles are: 1, 0001, A, a, I, and i.

.nr Hb 2 lowest heading level that is stand-alone (i. e., *not* run-in with the following text).

.nr Hc 0 lowest heading level that is centered.

.nr Hs 2 lowest heading level after which there is a blank line.

.nr Ht 0 heading marks will be concatenated.

.nr Hu 2 unnumbered headings (.HU) are equivalent to numbered headings at this level for spacing, font, and counting.

.nr Cl 2 lowest heading level to be saved for the table of contents.

.nr Ej 0 lowest heading level that forces the start of a new page.

Default Heading Style	
to get:	*type:*
n. HEADING Text ...	.H 1 "HEADING" Text ...
n.n Heading Text ...	.H 2 "Heading" Text ...
n.n.n Heading. Text ...	.H 3 "Heading." Text ...

Lists and List Types

All lists have a *list begin* macro, one or more *list items*—each consisting of a .LI macro followed by the *list item text*—and the *list end* macro .LE. That is, lists are typed like this:

list begin macro
.LI
list item text ...
.LI
list item text ...
⋮
.LE

where the *list begin macro* is one of the following:

.AL *[type] [indent]*	automatic list (*type* is 1, A, a, I, or i; if omitted, defaults to 1)
.BL *[indent]*	bullet list
.DL *[indent]*	dash list
.ML *mark [indent]*	marked list (*mark* is the desired mark)
.RL *[indent]*	reference list
.VL *indent*	variable list

indent is the number of characters of indentation (from the current indent) at which the list is to start; if it is optional and omitted, the default indentation for the given list style is used; *mark* will appear to the left of the indentation.

☛ *The output for the following is shown on p. 5.*

```
.AL 1
.LI
Pencilpusher, I., and Hardwired, X.
A New Kind of Set Screw.
.I "Proc. IEEE"
.B 75
(1976), 235-41.
.LI
Nails, H., and Irons, R.
Fasteners for Printed Circuit Boards.
.I "Proc. ASME"
.B 123
(1974), 23-24.
.LE
```

1. Pencilpusher, L., and Hardwired, X. A New Kind of Set Screw. *Proc. IEEE* 75 (1976), 235-41.

2. Nails, H., and Irons, R. Fasteners for Printed Circuit Boards. *Proc. ASME* 123 (1974), 23-24.

Nested Lists

```
This is ordinary text to show
the margins of the page.
.AL 1
.LI
First-level item.
.AL a
.LI
Second-level item.
.LI
Another second-level item, but
somewhat longer.
.LE
.LI
Return to previous list (and to previous value
of indentation) at this point.
.LI
Another line.
.LE
.P
Now we're out of the lists and at the margin that
existed at the beginning of this example.
```

This is ordinary text to show the margins of the page.

1. First-level item.
    a. Second-level item.
    b. Another second-level item, but somewhat longer.
2. Return to previous list (and to previous value of indentation) at this point.
3. Another line.

Now we're out of the lists and at the margin that existed at the beginning of this example.

Italic, Bold, and Underlining

In the examples on pp. 4 and 7, the macros .I, .B, and .R are used to change to, respectively, the italic, bold, and roman fonts in *troff*. In *nroff*, both .I and .B cause underlining until the occurrence of .R, which turns it off. A single argument given to either .I or .B results in that argument being underlined by *nroff*, or printed in the corresponding font by *troff*.

Displays

Displays are blocks of text that are to be kept together—not split across pages. A static display (.DS) appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of a page if a static display is too big to fit there. A floating display (.DF), on the other hand, will "float" through the input text to the top of the next page if there is not enough room for it on the current page; thus, the text that *follows* a floating display in the input may *precede* it in the output. Displays can be positioned at the left margin, indented, or centered.

.DS *[format] [fill]*	.DF *[format] [fill]*
text ...	*text* ...
.DE	.DE

where *format* and *fill* have the following meanings:

format		*fill*	
Code	*Meaning*	*Code*	*Meaning*
""	no indent	""	no fill
0	no indent	0	no fill
1	indent	1	fill
2	center		

```
Highland Avenue, Mountain Station,
South Orange, Maplewood, Millburn, Short Hills;
.DS 1
and now
for something
completely different
.DE
Summit, Chatham, Madison,
Convent Station, Morristown, New Providence,
Murray Hill, Berkeley Heights.
```

Highland Avenue, Mountain Station, South Orange, Maplewood, Millburn, Short Hills;

and now
for something
completely different

Summit, Chatham, Madison, Convent Station, Morristown, New Providence, Murray Hill, Berkeley Heights.

Footnotes

Two styles of footnote marking are shown on p. 7. In the first, the asterisk is the mark placed on the footnote and the following .FS macro call, while in the second, a number is *automatically* generated to mark the footnote. The macros .FS and .FE are used to delimit the footnote text that is to appear at the bottom of the page.

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without this basic tool,*
.FS *
As first shown by Tiger & Leopard (1975).
.FE
few assemblies could be completed.
They may lack the popular\*F
.FS
According to Panther & Lion (1977).
.FE
appeal of the sledgehammer . . .
```

Among the most important occupants of the workbench are the long-nosed pliers. Without this basic tool,* few assemblies could be completed. They may lack the popular[1] appeal of the sledgehammer ...

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

1. According to Panther & Lion (1977).

Simple Letter—Example

☛ *The output for the following is shown on p. 8.*

```
.nr Pt 0
.ND "May 1, 1977"
.TL
PWB/MM Class
.AU "J. J. Jones" JJJ PY 9999 5001 1Q-100
.MT ""
.DS
To All Students:
.DE
.P
There will be a class on the document preparation
facilities of PWB/MM on November 15-18.
This class lasts for 4 half-day (morning) sessions,
each consisting of a lecture
and practice exercises on the system.
.P
The meeting rooms for the class are:
.DS 1
.ta 15n                    (n represents character positions)
Monday—4D-502                        (— indicates a tab)
Tuesday—4D-502
Wednesday—2B-639
Thursday—2C-641.
.DE
.P
Please read the following before attending class:
.DL
.LI
.I "UNIX for Beginners,"
Sections I and II.
.LI
.I
A Tutorial Introduction to the UNIX Text Editor.
.R
.LE
```

(input example continued on the next page)

Bell Laboratories

subject: **PWB/MM Class** date: **May 1, 1977**

from: **J. J. Jones**
PY 9999
1Q-100 x5001

To All Students:

There will be a class on the document preparation facilities of PWB/MM on November 15-18. This class lasts for 4 half-day (morning) sessions, each consisting of a lecture and practice exercises on the system.

The meeting rooms for the class are:

Monday	4D-502
Tuesday	4D-502
Wednesday	2B-639
Thursday	2C-641.

Please read the following before attending class:

- *UNIX for Beginners*, Sections I and II.
- *A Tutorial Introduction to the UNIX Text Editor.*

These can be obtained from the Computing Information Library.

PY-9999-JJJ-ae **J. J. Jones**

Copy to
G. H. Hurtz
S. P. LeName

(input example continued from the previous page)

```
.P
These can be obtained from the Computing
Information Library.
.SG ae
.NS
G. H. Hurtz
S. P. LeName
.NE
```

Technical Memorandum—Example

☛ *The output for the following is shown on pp. 10-12.*

```
.nr Pt 1
.ND "June 29, 1977"
.TL 12345 666666
On Constructing a Table of All
Even Prime Numbers
.AU "S. P. LeName" SPL PY 9999 4000 1Z-123
.AU "G. H. Hurtz" GHH PY 9999 4001 1Z-121
.TM 76543210
.AS
.P
This is an abstract for a technical memorandum.
The abstract will appear on the cover
sheet and on the first page
.I unless
the macro .AS has an argument of 1, in which case
the abstract will be printed only on the cover sheet.
The TM number appears on the cover sheet
and on the first page.
"Other Keywords" appear only on the cover sheet.
.P
The abstract may consist of one or more paragraphs;
it must fit on the cover sheet.
.AE
.OK "Prime Numbers" Even
.MT
.H 1 "INTRODUCTORY MATERIAL"
The first line of the body of the memorandum
immediately follows the macro call for
the heading (.H).
Alternately, lower-level heading macros may follow it,
as well as macros for lists, paragraphs, and so on.
A brief example of a list follows:
.AL A
.LI
This is the first item in an alphabetical
list in the body of this memorandum.
.LI
This is the second item in the list.
.AL 1
.LI
This is the first item in a (numbered) sub-list.
.LI
This is the second item in that sub-list.
.LE
.LE
.P
This is the second paragraph under the first heading.
In addition to alphabetized and numbered lists, there
are bullet lists, dash lists, variable lists, etc.
.H 2 "First Second-Level Heading"
This is the first paragraph under a
second-level heading.
Notice how that heading is numbered and
where the heading and text are printed.
.H 1 "SECOND FIRST-LEVEL HEADING"
This is the first paragraph under the
second first-level heading of the memorandum.
```

(input example continued on the next page)

Bell Laboratories

subject: On Constructing a Table of All Even Prime Numbers
Case: 12345
File: 666666

date: June 29, 1977

from: S. P LeName
PY 9999
1Z-123 x4000

G. H. Hurtz
PY 9999
1Z-121 x4001

TM: 76543210

ABSTRACT

This is an abstract for a technical memorandum. The abstract will appear on the cover sheet and on the first page *unless* the macro .AS has an argument of 1, in which case the abstract will be printed only on the cover sheet. The TM number appears on the cover sheet and on the first page. "Other Keywords" appear only on the cover sheet.

The abstract may consist of one or more paragraphs; it must fit on the cover sheet.

MEMORANDUM FOR FILE

1. INTRODUCTORY MATERIAL

The first line of the body of the memorandum immediately follows the macro call for the heading (.H). Alternately, lower-level heading macros may follow

---

(input example continued from the previous page)

```
.HU REFERENCES
.RL
.LI
Pencilpusher, I., and Hardwired, X.
A New Kind of Set Screw.
.I "Proc. IEEE"
.B 75
(1976), 235-41.
.LI
Nails, H., and Irons, R.
Fasteners for Printed Circuit Boards.
.I "Proc. ASME"
.B 123
(1974), 23-24.
.LE
.SG rfg
.NS 3
.NS 2
G. B. Brown
C. P. Jones
J. J. Smith
.NE
.CS 2 1 3 0 0 2
```

11

it, as well as macros for lists, paragraphs, and so on. A brief example of a list follows:

A. This is the first item in an alphabetical list in the body of this memorandum.

B. This is the second item in the list.

1. This is the first item in a (numbered) sub-list.
2. This is the second item in that sub-list.

This is the second paragraph under the first heading. In addition to alphabetized and numbered lists, there are bullet lists, dash lists, variable lists, etc.

1.1 **First Second-Level Heading**

This is the first paragraph under a second-level heading. Notice how that heading is numbered and where the heading and text are printed.

2. **SECOND FIRST-LEVEL HEADING**

This is the first paragraph under the second first-level heading of the memorandum.

REFERENCES

[1] Pencilpusher, L., and Hardwired, X. A New Kind of Set Screw. *Proc. IEEE* 75 (1976), 235-41.

[2] Nails, H., and Irons, R. Fasteners for Printed Circuit Boards. *Proc. ASME* 123 (1974), 23-24.

S. P. LeName

PY-9999-SPL/GHH-rfg G. H. Hurtz

Att.

Copy (without att.) to
G. B. Brown
C. P. Jones
J. J. Smith

12

Bell Laboratories Cover Sheet for TM

The information contained herein ... not for publication ...

Title: **On Constructing a Table of All Even Prime Numbers** Date: **June 29, 1977**

TM: **76543210**

Other Keywords: **Prime Numbers**
Even

Author(s)	Location	Ext.	Charging Case: 12345
S. P. LeName	PY 1Z-123	4000	Filing Case: 666666
G. H. Hurtz	PY 1Z-121	4001	

ABSTRACT

This is an abstract for a technical memorandum. The abstract will appear on the cover sheet and on the first page *unless* the macro .AS has an argument of 1, in which case the abstract will be printed only on the cover sheet. The TM number appears on the cover sheet and on the first page. "Other Keywords" appear only on the cover sheet.

The abstract may consist of one or more paragraphs; it must fit on the cover sheet.

Pages Text: 2 Other: 1 Total: 3

No. Figures: 0 No. Tables: 0 No. Refs.: 2

Z-0000-X SEE REVERSE SIDE FOR DISTRIBUTION LIST

Memorandum-Style Macros

Macros for a memorandum-style document must be invoked in the order shown on pp. 9-10. Once the "memorandum type" (.MT) macro has been invoked, none of the macros that precede it can be used. The .MT macro controls the format of the "subject, date, from" portion of the first page of the memorandum. Different arguments to the .MT macro will produce different kinds of memoranda:

Code	*Meaning*
.MT ""	no memorandum type is printed
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER'S NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Released-Paper style
.MT 5	External Letter

The input and the resulting output for a simple letter are shown on pp. 7-8. Note that the .TM, .AS/.AE, and .OK macros are *not* used there, and that the .MT macro has a *null* argument (""). Documents of the type shown on pages 2-3 (essentially plain text) are produced by omitting, as well, the other "memorandum-style" macros: .ND, .TL, .AU, and .MT at the beginning of the document, and .SG, .NS/.NE, and .CS at the end.

Like the .MT macro, the notation macro (.NS) may also take different arguments to produce a variety of notations following the signature line:

Code	*Meaning*
.NS ""	Copy to
.NS 0	Copy to
.NS	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to

If the .CS macro is included in the input file (see last line of p. 10) *and* if the -rB2 option is included on the command line (see p. 16), a cover sheet is generated (see p. 12). (The 6 arguments to .CS are the data for the bottom of the TM cover sheet: "Pages Text," "Other," etc.) Similarly, the .TC macro, together with the -rB1 or -rB3 option (see p. 16) generates a table of contents; .CS and .TC can occur only at the end of a document.

Two-Column Output

```
.DS 2
The Declaration of Independence
.DE
.2C
.P
When in the Course of human events, it becomes
necessary for one people to dissolve the political
bands 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.
.P
We hold these truths to be self-evident, that all men
are created equal, ...
```

The Declaration of Independence

When in the Course of human events, it becomes necessary for one people to dissolve the political bands 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, ...

Equations

A stand-alone equation is built within a display.

```
.DS 2
.EQ
x sup 2 over a sup 2 ~=~ sqrt { pz sup 2 + qz + r }
.EN
.DE
```

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r}$$

```
.DS 1
.EQ
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 ] times left [ pile { alpha
above beta above gamma } right ]
.EN
.DE
```

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

In-line equations may appear in running text if a character has been defined to mark the left and right ends of the equation. Normally, $ is used as that character and is so defined by typing the following three lines at the beginning of the document:

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

```
The quantities $a dot$, $b dotdot$, $xi tilde times
y vec$ are the values that show ...
```

The quantities $\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$ are the values that show ...

This facility can be used for preparing text that contains subscripts and superscripts:

```
The quantity $ a sub j sup 3 $ is ...
```

The quantity a_j^3 is ...

For more examples, see p. 15 and Reference 4.

Tables

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

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

Global table options are *center, expand, box, allbox, doublebox,* and *tab (x).*

```
.DS
.TS
allbox ;
ci s s
c c c
n n n .
AT&T Common Stock
Year—Price—Dividend
1973—46-55—2.87
4—40-53—3.24
5—45-52—3.40
6—51-59—.95*
.TE
* (first quarter only)
.DE
```

AT&T Common Stock

Year	Price	Dividend
1973	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

(— indicates a tab)

```
.EQ
delim $$
.EN
.DS
.TS
box ;
cf2 cf2
l l .
Name—Definition
—
.sp
Sine—$sin ( x ) = 1 over 2j ( e sup jx - e sup -jx )$
Zeta—$zeta ( s ) = \
sum from k=1 to inf k sup -s ~~( Re~s > 1 )$
.TE
.DE
```

Name	*Definition*
Sine	$\sin(x)=\frac{1}{2j}(e^{jx}-e^{-jx})$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s} \quad (\mathrm{Re}\ s>1)$

For more examples, see Reference 3.

How to Get Output

Documents with text only:

nroff: mm *[options]* files
or nroff *[options]* -mm files
troff: troff *[options]* -mm files

Text and equations:

nroff: mm -e *[options]* files
or neqn files | nroff *[options]* -mm -
troff: eqn files | troff *[options]* -mm -

Text and tables:

nroff: mm -t *[options]* files
or tbl files | nroff -mm *[options]* -
troff: tbl files | troff -mm *[options]* -

Text, tables, and equations:

nroff: mm -t -e *[options]* files
or tbl files | neqn | nroff *[options]* -mm -
troff: tbl files | eqn | troff *[options]* -mm -

The following options may be specified on the above PWB/UNIX shell command lines:

-o*k,m-n*	print only page *k*, and pages *m* through *n.*
-rB1	include macros for the table of contents.
-rB2	include macros for the cover sheet.
-rB3	include macros for both.
-rC1	*OFFICIAL FILE COPY* in footer.
-rC2	*DATE FILE COPY* in footer.
-rC3	*DRAFT* in footer.
-rL*n*	set page length to *n* lines.*
-rN1	page header at *bottom* of first page only.
-rN2	no page number on first page.
-rN3	*section-page* numbering.
-rO*n*	set page offset to *n* characters.*
-rW*n*	set line width to *n* characters.*

Terminal type and/or pitch are usually indicated by the -hp, -ti, -450, -300S, and/or -12 options of the *mm*(I) command, if it is used (see Reference 6); otherwise, they are specified by one of the *nroff* -T*name* options.

References

1. *PWBMM—Programmer's Workbench Memorandum Macros* by D. W. Smith and J. R. Mashey.
2. *A Tutorial Introduction to the UNIX Text Editor* by B. W. Kernighan.
3. *Tbl—A Program to Format Tables* by M. E. Lesk.
4. *Typesetting Mathematics—User's Guide (Second Edition)* by B. W. Kernighan and L. L. Cherry.
5. *NROFF/TROFF User's Manual* by J. F. Ossanna.
6. *PWB/UNIX User's Manual—Edition 1.0* by T. A. Dolotta, R. C. Haight, and E. M. Piskorik, eds.

* For *nroff, n* must be an *unscaled* number representing lines or character positions. For *troff, n* must be *scaled.*

The PWB/UNIX* document entitled:

PWB/MM Tutorial

is not yet available.

* UNIX is a Trademark/Service Mark of the Bell System.

Tbl — A Program to Format Tables

M. E. Lesk

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. It is available on the PDP-11 UNIX* system and on Honeywell 6000 GCOS. 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:

1970 Federal Budget Transfers
(in billions of dollars)

State	Taxes collected	Money spent	Net
New York	22.91	21.35	−1.56
New Jersey	8.33	6.96	−1.37
Connecticut	4.12	3.10	−1.02
Maine	0.74	0.67	−0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

September 4, 1977

* UNIX is a Trademark/Service Mark of the Bell System

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of commands) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. 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* [3] or various layout macro packages [4,5,6], 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 [4]) 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:

```
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.

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:

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.

The *tbl* program tries to keep boxed tables on one page by issuing appropriate "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) 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 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* (one en is about the width of the letter 'n').* 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.

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

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 tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. 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.

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.

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.

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 in the option *tabs* option. 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, space within a table may be produced by ".sp" commands in the data.

Full width horizontal lines — An input *line* containing only the character _ (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* cc ntaining only the character _ or = is taken to be a single or double line extending the full width of the *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.

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 10 for an illustration of included text blocks in a table. If more than twenty or thirty 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* itself, 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 and font, using the **p** 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.

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 9 and 12. 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.

Usage.

On UNIX, *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* memorandum layout commands as well as tables, the normal command would be

```
tbl file-1 file-2 . . . | eqn | troff −ms
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables.

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.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff*, producing the 'too many number registers' message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms #*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.

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 l.
Language ⓣ Authors ⓣ Runs on

Fortran ⓣ Many ⓣ Almost anything
PL/1 ⓣ IBM ⓣ 360/370
C ⓣ BTL ⓣ 11/45,H6000,370
BLISS ⓣ Carnegie-Mellon ⓣ PDP-10,11
IDS ⓣ Honeywell ⓣ H6000
Pascal ⓣ Stanford ⓣ 370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
Year ⓣ Price ⓣ Dividend
1971 ⓣ 41-54 ⓣ $2.60
2 ⓣ 41-54 ⓣ 2.70
3 ⓣ 46-55 ⓣ 2.87
4 ⓣ 40-53 ⓣ 3.24
5 ⓣ 45-52 ⓣ 3.40
6 ⓣ 51-59 ⓣ .95*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (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:

```
.TS
box;
cfB 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
```

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
Era ⓣ Formation ⓣ Age (years)
Precambrian ⓣ Reading Prong ⓣ > 1 billion
Paleozoic ⓣ Manhattan Prong ⓣ 400 million
Mesozoic ⓣ T{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T} ⓣ 200 million
Cenozoic ⓣ Coastal Plain ⓣ T{
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.
Name ⓣ Definition
.sp
.vs +2p
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)$
.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	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s}\ \ (\mathrm{Re}\ s>1)$

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 ⓣ 244
Tube ⓣ 66
Sub-surface ⓣ 22
Surface ⓣ 156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
Journeys ⓣ 674 million
Average length ⓣ 4.55 miles
Passenger miles ⓣ 3,066 million
.T&
l r
a r.
Passenger traffic \- road
Journeys ⓣ 2,252 million
Average length ⓣ 2. 26 miles
Passenger miles ⓣ 5,094 million
.T&
l n
a n.
.sp .5
Vehicles ⓣ 12,521
Railway motor cars ⓣ 2,905
Railway trailer cars ⓣ 1,269
Total railway ⓣ 4,174
Omnibuses ⓣ 8,347
.T&
l n
a n.
.sp .5
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
.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:

```
.ps 8
.vs 10p
.TS
center box;
c s s
ci s s
c c c
lB l n.
New Jersey Representatives
(Democrats)
.sp .5
Name ⓣ Office address ⓣ Phone
.sp .5
James J. Florio ⓣ 23 S. White Horse Pike, Somerdale 08083 ⓣ 609-627-8222
William J. Hughes ⓣ 2920 Atlantic Ave., Atlantic City 08401 ⓣ 609-345-4844
James J. Howard ⓣ 801 Bangs Ave., Asbury Park 07712 ⓣ 201-774-1600
Frank Thompson, Jr. ⓣ 10 Rutgers Pl., Trenton 08618 ⓣ 609-599-1619
Andrew Maguire ⓣ 115 W. Passaic St., Rochelle Park 07662 ⓣ 201-843-0240
Robert A. Roe ⓣ U.S.P.O., 194 Ward St., Paterson 07510 ⓣ 201-523-5152
Henry Helstoski ⓣ 666 Paterson Ave., East Rutherford 07073 ⓣ 201-939-9090
Peter W. Rodino, Jr. ⓣ Suite 1435A, 970 Broad St., Newark 07102 ⓣ 201-645-3213
Joseph G. Minish ⓣ 308 Main St., Orange 07050 ⓣ 201-645-6363
Helen S. Meyner ⓣ 32 Bridge St., Lambertville 08530 ⓣ 609-397-1830
Dominick V. Daniels ⓣ 895 Bergen Ave., Jersey City 07306 ⓣ 201-659-7700
Edward J. Patten ⓣ Natl. Bank Bldg., Perth Amboy 08861 ⓣ 201-826-4610
.sp .5
.T&
ci s s
lB l n.
(Republicans)
.sp .5v
Millicent Fenwick ⓣ 41 N. Bridge St., Somerville 08876 ⓣ 201-722-8200
Edwin B. Forsythe ⓣ 301 Mill St., Moorestown 08057 ⓣ 609-235-6622
Matthew J. Rinaldo ⓣ 1961 Morris Ave., Union 07083 ⓣ 201-687-4235
.TE
.ps 10
.vs 12p
```

Output:

New Jersey Representatives

Name	Office address	Phone
	(Democrats)	
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
	(Republicans)	
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

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.
Bell Labs Locations
Name ⓣ Address ⓣ Area Code ⓣ Phone
Holmdel ⓣ Holmdel, N. J. 07733 ⓣ 201 ⓣ 949-3000
Murray Hill ⓣ Murray Hill, N. J. 07974 ⓣ 201 ⓣ 582-6377
Whippany ⓣ Whippany, N. J. 07981 ⓣ 201 ⓣ 386-3000
Indian Hill ⓣ Naperville, Illinois 60540 ⓣ 312 ⓣ 690-2000
.TE
```

Output:

Bell Labs Locations

Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

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

_
Name⓪ Description⓪ Practical Information

_
T{
American Museum of Natural History
T}⓪T{
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}⓪ Hours⓪ 10-5, ex.  Sun 11-5, Wed. to 9
\^⓪\^⓪ Location⓪T{
Central Park West & 79th St.
T}
\^⓪\^⓪ Admission⓪ Donation: $1.00 asked
\^⓪\^⓪ Subway⓪ AA to 81st St.
\^⓪\^⓪ Telephone⓪ 212-873-4225

_
Bronx Zoo⓪T{
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}⓪ Hours⓪T{
10-4:30 winter, to 5:00 summer
T}
\^⓪\^⓪ Location⓪T{
185th St. & Southern Blvd, the Bronx.
T}
\^⓪\^⓪ Admission⓪ $1.00, but Tu,We,Th free
\^⓪\^⓪ Subway⓪ 2, 5 to East Tremont Ave.
\^⓪\^⓪ Telephone⓪ 212-933-1759

_
Brooklyn Museum⓪T{
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}⓪ Hours⓪ Wed-Sat, 10-5, Sun 12-5
\^⓪\^⓪ Location⓪T{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^⓪\^⓪ Admission⓪ Free
\^⓪\^⓪ Subway⓪ 2,3 to Eastern Parkway.
\^⓪\^⓪ Telephone⓪ 212-638-5000

_
T{
New-York Historical Society
T}⓪T{
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}⓪ Hours⓪T{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^⓪\^⓪ Location⓪T{
Central Park West & 77th St.
T}
\^⓪\^⓪ Admission⓪ Free
\^⓪\^⓪ Subway⓪ AA to 81st St.
\^⓪\^⓪ Telephone⓪ 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, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: $1.00 asked AA to 81st St. 212-873-4225
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	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. $1.00, but Tu,We,Th free 2, 5 to East Tremont Ave. 212-933-1759
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	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
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-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

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 J. F. Ossanna, whose assistance with this program in particular has 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.

References.

[1] J. F. Ossanna, *NROFF/TROFF User's Manual*, Computing Science Technical Report No. 55, Bell Laboratories, 1976.

[2] K. Thompson and D. M. Ritchie, "The UNIX Time-Sharing System," *Comm. ACM.* 17, pp. 365–75 (1974).

[3] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. ACM.* 18, pp. 151–57 (1975).

[4] M. E. Lesk, *Typing Documents on UNIX*, Bell Laboratories internal memorandum.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX*, Computing Science Technical Report No. 44, Bell Laboratories, July 1976.

[6] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros*, Bell Laboratories memorandum.

List of Tbl Command Characters and Words

Command	*Meaning*	*Section*
a A	Alphabetic subcolumn	2
allbox	Draw box around all items	1
b B	Boldface item	2
box	Draw box around table	1
c C	Centered column	2
center	Center table in page	1
doublebox	Doubled box around table	1
e E	Equal width columns	2
expand	Make table full line width	1
f F	Font change	2
i I	Italic item	2
l L	Left adjusted column	2
n N	Numerical column	2
nnn	Column separation	2
p P	Point size change	2
r R	Right adjusted column	2
s S	Spanned item	2
t T	Vertical spanning at top	2
tab (*x*)	Change data separator character	1
T{ T}	Text block	3
w W	Minimum width value	2
.*xx*	Included *troff* command	3
|	Vertical line	2
||	Double vertical line	2
^	Vertical span	2
\^	Vertical span	3
=	Double horizontal line	2,3
_	Horizontal line	2,3

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

troff is a text-formatting program for driving the Graphic Systems phototypesetter on the UNIX and GCOS operating systems. This device is capable of producing high quality text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. 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. It assumes that the reader is familiar with a formatter like **roff** on UNIX or GCOS. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

troff [1] is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX and GCOS operating systems. 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 two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

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' [4] and PWB/MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This memo was prepared with '−ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX and GCOS, and for 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:

The **troff** described here is the C-language version running on UNIX at Murray Hill, as documented in [1].

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. (Readers who use **roff** will find the approach familiar.) 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. There are 15 point sizes, listed 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 one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. 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 in 9 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 runs on a PDP-11/45

type

\s8UNIX\s10 runs 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. 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 "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

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 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

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 other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells **troff** what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate .fp commands should appear at the beginning of your document if you do not use the standard fonts.

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. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in [1].

Special characters have four-character names beginning with \(, and 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

$$\Sigma(\alpha \times \beta) \rightarrow \infty$$

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	∞

A complete list of these special names occurs in Appendix A.

In **eqn** [2] 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 get a backslash printed, 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, as in

```
.ll 6i
```

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by 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
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don't* use **troff** directly; use the **tbl** program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

```
  1          2          3
 40         50         60
700        800        900
```

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...\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 quite analogous to \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. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so **eqn** replaces this by

```
>\h'-0.3m'>
```

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 (1/432 inch). 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.

As a live example of this kind of construction, all of the command names in the text, like .sp, were done by overstriking with a slight offset. The commands for .sp are

```
.sp\h'-\w'.sp'u'\h'1u'.sp
```

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp' again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen \0, which 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

```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which makes

système téléphonique

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 entirely analogous, 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 é 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.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command .ds. The line

```
.ds e \o"e\'"
```

defines the string e to have the value \o"e\'"

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 téléphone, given the definition of the string e as above, we can say t*el*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
..
.de BE          \" end indented block
.sp
.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
```

In **roff**, one can say

```
.he 'left top'center top'right top'
.fo 'left bottom'center bottom'right bottom'
```

to get headers and footers automatically on every page. Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

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, like this:

```
.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 4.) 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 [4], 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 R           \" font
.sp 0.5v        \" half a line
.ti +3m
..
```

This sets the font to Roman 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 (1/432 inch) 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 TROFF
```

will produce TROFF.

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 TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

produces (TROFF). 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 the one used to make the 'bold roman' we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'-\w'\\$1'u+1u'\\$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 .SH which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "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 .SH macro:

```
.nr SH 0        \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1     \" increment number
.ps \\n(PS-1        \" decrease PS
\\n(SH. \\$1        \" number. title
.ps \\n(PS          \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used \\n(SH instead of \n(SH and \\n(PS instead of \n(PS. If we had used \n(SH, 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 (as on the top of this page), but a user could

supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .SH 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 .SH 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(SH=1 .sp 2i        \" first 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 SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=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(SH>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 R
.ps 10
... any other processing ...
.ev            \" return to previous 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 the most 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 doesn't fit
.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.

Acknowledgements

I am deeply indebted to 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.

References

[1] J. F. Ossanna, *NROFF/TROFF* User's Manual, Bell Laboratories internal memorandum.

[2] B. W. Kernighan, *A System for Typesetting Mathematics — User's Guide (Second Edition)*, Bell Laboratories internal memorandum.

[3] M. E. Lesk, *TBL — A Program to Format Tables*, Bell Laboratories internal memorandum.

[4] M. E. Lesk, *Typing Documents on UNIX*, Bell Laboratories internal memorandum.

[5] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros*, Bell Laboratories internal memorandum.

Appendix A: Phototypesetter Character Set

These characters exist in roman, italic, and bold. To get the one on the left, type the four-character name on the right.

ff	\(ff	fi	\(fi	fl	\(fl	ffi	\(Fi	ffl	\(Fl
‗	\(ru	—	\(em	¼	\(14	½	\(12	¾	\(34
©	\(co	°	\(de	†	\(dg	′	\(fm	¢	\(ct
®	\(rg	•	\(bu	□	\(sq	-	\(hy		

(In bold, \(sq is ■.)

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	Ⓒ	\(bs
§	\(sc	‡	\(dd	◥	\(lh	◤	\(rh
⌈	\(lt	⌉	\(rt	⌈	\(lc	⌉	\(rc
⌊	\(lb	⌋	\(rb	⌊	\(lf	⌋	\(rf
{	\(lk	}	\(rk	|	\(bv	ς	\(ts
|	\(br	|	\(or	_	\(ul	‾	\(rn
•	\(bu						

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ` is the other quote mark.

´	\'	`	\`	–	\-	_	_

These characters exist only on the special font, but they do not have four-character names:

" { } < > ~ ^ \ # @

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
$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$

A B G D E Z Y H I K L M N C O P R S T U F X Q W
$A\ B\ \Gamma\ \Delta\ E\ Z\ H\ \Theta\ I\ K\ \Lambda\ M\ N\ \Xi\ O\ \Pi\ P\ \Sigma\ T\ Y\ \Phi\ X\ \Psi\ \Omega$

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan

Bell Laboratories,
Murray Hill, New Jersey 07974

Lorinda L. Cherry

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX and GCOS operating systems.

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

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

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

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

can be learned in an hour or so.

The language interfaces directly with the phototypesetting 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.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan

Bell Laboratories,
Murray Hill, New Jersey 07974

Lorinda L. Cherry

Bell Laboratories,
Murray Hill, New Jersey 07974

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. 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 works as a preprocessor for the typesetter formatter, TROFF[1], 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.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 teletypes. The input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark 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 the TROFF and NROFF macro package package '−ms' developed by M. E. Lesk[3], which allows 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

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

There is alsc 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 19.

3. Input spaces

Spaces 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. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

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 *very* 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 it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like \(*bs* for the Bell System sign ⓑ.

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 ordinary spaces (or 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 if necessary:

```
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 14.

9. 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 which

decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, *use braces* to make clear what goes with what.

10. 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 lousy, 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:

$$(a^2/b_2)^{1/2}$$

which is

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

11. 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

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

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically 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 14 bold x = y +
  size 14 {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 12 { ... }
```

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. 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 one character TROFF name or number for the font. 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 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 thoughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

‡Like this one, in which we have a few random expressions like x_i and π^2. The sizes for these were set by the command *gsize −2*.

13. Diacritical Marks

To get funny 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	$\bar{x}$
x under	$\underline{x}$

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.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. 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 place-holder 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 "\"". TROFF characters like *\(bs* can appear unquoted, but more complicated things like horizontal and vertical motions with *\h* and *\v* should always be quoted. (If you've never heard of *\h* and *\v*, ignore this section.)

15. 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 at all 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
```

isn't going to work, because there isn't room for the $x+y$ part after the *mark* remembers where the x is.

16 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 the are not likely to look very good. One exception is the *floor* and *ceiling* characters:

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

produces

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

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.

17. 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.

18. Matrices

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

$$x,\ x^2$$
$$y,\ y^2$$

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 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.

19. Shorthand for In-line Equations

In a mathematical document, 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. Although this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '-ms', .EQ and .EN imply a displayed equation.

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 the primary variable, and let $beta$ be zero. Then we can show that $x sub 1$ 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 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.

20. 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 sub 1 + y sub i sub 1
```

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

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

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 ' / '
```

If you need different things to print on a terminal and on the typesetter, it is sometimes worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine*, the definition only applies for EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, 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 (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n*. 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.

As an example of local motions, consider tucking the limits in under an integral sign. Normally if you say

```
int sub 0 sup 1
```

it looks like

$$\int_0^1$$

which is awful. The intuitively appealing

```
int from 0 to 1
```

is

$$\int_0^1$$

which is not normally used. But if you say

```
int sub back 40 down 50 0 sup up 30 1
```

you get

$$\int_0^1$$

(These values are determined experimentally.) Of course this is a nuisance to type, so you would first make definitions like this:

```
tdefine lower ' sub back 40 down 50 '
tdefine upper ' sup up 30 '
```

and then say

```
int lower 0 upper 1
```

22. 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
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
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.

>=	≥
<=	≤
==	≡
!=	≠
+-	±
->	→
<-	←
<<	≪
>>	≫
inf	∞
partial	∂
half	½
prime	′
approx	≈
nothing	
cdot	·
times	×
del	∇

grad	∇
...	$\ldots$
,....,	$,\ldots,$
sum	Σ
int	$\int$
prod	Π
union	$\cup$
inter	$\cap$

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	o
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		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~, ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. 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 between lines x and y, file z

where *x* and *y* are approximately the lines between which the trouble occurred, and *z* is the name of the file in question. The line numbers are 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 (on UNIX only),

```
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* (on GCOS, use *./checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a 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 an even bigger buffer. 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. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

```
eqn files | troff
```

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

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

To run the same document on the GCOS typesetter, use

```
eqn files | troff -g (other options) | gcat
```

A compatible version of EQN can be used on devices like teletypes and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 teletype, for example, use

```
neqn files | nroff
```

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

```
neqn files | nroff -Tx
```

where *x* is the terminal type you are using, such as *300* or *300S*.

EQN and NEQN can be used with the TBL program[2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

```
tbl files | eqn | troff
tbl files | neqn | nroff
```

26. Use on GCOS

This space intentionally left blank

27. Acknowledgments

We are deeply indebted to 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] J. F. Ossanna, "TROFF User's Manual", Bell Laboratories internal memorandum.

[2] M. E. Lesk, "Typing Documents on UNIX", Bell Laboratories internal memorandum.

[3] M. E. Lesk, "TBL— A Program for Setting Tables", Bell Laboratories internal memorandum.

New Graphic Symbols for EQN and NEQN

Carmela Scrocca

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

There is now available on UNIX and GCOS a set of special characters frequently used in technical typing. In the past, authors have sometimes written out these symbols in English; others just assumed their secretary or typist had these symbols ready and waiting. These characters, however, are not part of the standard terminal or typesetter character sets, but are built-up of those already available. They can presently be produced for phototypesetter output by using EQN/TROFF; NEQN/NROFF can be used for computer terminal output.

This document displays these characters, shows how to use them, and discusses what is involved in making a special character.

New Graphic Symbols for EQN and NEQN

Carmela Scrocca

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

There is now available on UNIX and GCOS a set of special characters frequently used in technical typing. These characters supplement the ones that come with the typesetter and terminal which both have their own set of standard characters. These special characters are accessed through the math typesetting programs NEQN and EQN.[1] Processed through the NROFF and TROFF[2] formatting programs, the characters can be output on either a computer terminal or a phototypesetter. Using various NROFF/TROFF conventions, two or more of these existing characters can be built-up and pieced together to draw a new character.

Sections 1 and 2 of this document give a list of these characters and tell how to access and use them. In Sections 3 and 4, the reader will see what is involved in making a special character for both phototypesetter and computer terminal output.

1. The Characters

Table 1 gives a list of the characters, their meanings, and the names by which EQN recognizes them.

The user should be aware that these special characters are not built into the NEQN/EQN program, but are stored in directory /usr/pub; the filename is *eqnchar*. In order to use any of these symbols this file will have to be referenced. This can be done by using file *eqnchar* as the first filename in your output command, such as

neqn /usr/pub/eqnchar filenames | nroff.

for computer terminal output. For phototypesetter output use EQN/TROFF instead of NEQN/NROFF. On GCOS, the characters are in file ./eqnchar.

Some users will find a constant need for only a few of the special characters. It may be convenient for these users to take their few selected characters, and write them into a file in their own directory. Other users may only need a few characters for use in one particular document. They could edit /usr/pub/eqnchar, copy the desired character definitions into a separate file, and read that file into the beginning of their document file.

Appendices 1a and 1b provide additional characters with their corresponding EQN names. The characters list in Appendix 1a are from the phototypesetter character set and have been assigned EQN names. /usr/pub/eqnchar must be referenced in the output command to use characters in this set. Appendix 1b contains a list of characters already built into EQN and can be used directly with the program.

character name	character	EQN name
sum of two elements	$\oplus$	ciplus
product of two elements	$\otimes$	citimes
is congruent to	$\cong$	=wig
approximately equal to	$\doteq$	=dot
equals by definition	$\triangleq$	=del
large star	$\ast$	bigstar
centered star	$*$	star
or	$\vee$	orsign
and	$\wedge$	andsign
for all	$\forall$	oppA
there exists	$\exists$	oppE
is included in	$\sqsubset$	incl
not a member of	$\notin$	nomem
angstrom	$\mathring{A}$	angstrom
less than or approximately equal to	$\lesssim$	<wig
greater than or approximately equal to	$\gtrsim$	>wig
not less than	$\not<$	|<
not greater than	$\not>$	|>
left angle bracket	$\langle$	langle
right angle bracket	$\rangle$	rangle
hbar	$\hbar$	hbar
parallel	$\parallel$	||
perpendicular	$\perp$	ppd
angle	$\angle$	ang
right angle	$\llcorner$	rang
implies and is implied by	$\longleftrightarrow$	<->
implies and is implied by	$\Longleftrightarrow$	<=>
vertical ellipsis	$\vdots$	3dot
therefore	$\therefore$	thf

Table I. The Special Characters

2. Usage

The reader should be familiar with EQN as these characters work with the program and are used in the same manner as any other mathematical symbol. An EQN name must be separated from surrounding input (with spaces) in order to be recognized as a special character. For example, just as you would say

```
pi~=~sum x sup i
```

to get

$$\pi = \sum x'$$

you could say

x sub 3 ~=wig~ pi star y sub i pd

to give you

$$x_3 \cong \pi * y_{\perp}$$

of which =*wig*, *star* and *ppd* are a few of the new special characters. These symbols will work in both displayed and in-line equations. There are no bold or italic versions of the symbols.

3. Creating Special Characters for EQN

The symbols discussed here were created by taking pre-existing characters and piecing them together with the input and output conventions and escape sequences made available through NROFF and TROFF. Most commonly used here were the local horizontal and vertical motions, overstrike and zero width functions, and the point-size change function. These are used primarily for phototypesetter output; making characters for computer terminal output will be discussed in Section 4. The definitions of all the special characters are listed in Appendix 2.

Local horizontal and vertical motions are used to move a character up, down, or to the left or right depending on where you want to position your character. These motions are generated by the escape sequences \u, \d, \r, \v and \h. The motions are expressed in terms of ems; an em is approximately the width of the letter 'm'. By using ems, the amount of motion will always be in proportion to character size. The \u and \d sequences give vertical motions of ½ em up and down, respectively; \r gives an upward motion of 1 em. The \v (vertical motion) and \h (horizontal motion) escape sequences allow you to move any fraction of an em. The distance must be enclosed in ' marks and the direction of movement can be indicated by making it either positive or negative. For example, if you wanted a downward vertical motion of 3/10 of an em, you would say \v'.3m', and an upward vertical motion of 6/10 of an em would be \v'-.6m'. The same basic rules apply to horizontal motions where positive moves to the right, and negative to the left. The "is much greater than" symbol >> shows a simple horizontal motion:

>\h'-.3m'>

(>> is not a special character, but built into EQN; see Appendix 1b.)

The overstrike function \o simply overprints characters on top of another, centered on the widest character. This function was used to create the "sum of two elements" symbol ⊖ by saying

\o'\(pl\(ci'

where \(pl and \(ci are the escapes for + and O. The string of characters to be overstruck must be enclosed in ' marks. When using the overstrike function be sure not to use any motions or it will not work.

Similar to the overstrike function is the bracket building function \b. Instead of centering one character on the other, the bracket building function piles the characters vertically. The "angle brackets" were built-up using the \b function:

\s-3\b'\(sl\e'\s0 ⟨

\s-3\b'\e\(sl'\s0 ⟩

(\e and \(sl are the escapes for \ and /.) The first character in the string is positioned at the top of the pile, and on down with the last character at the bottom. \s is the escape sequence for a size change; we'll get to this in a while.

The zero width function \z enables you to print a character without moving after it is printed. \z often makes it easier to position your next character rather than figuring out where the first one moved you to. This function is used with \z*n* where *n* is the character to be printed. \z can only be applied to one character at a time so the sequence would have to be repeated. An example of this can be the definition of the "vertical ellipsis" ⋮ where \z forces the dots to remain in place; otherwise, horizontal motions would be needed to realign the dots.

```
\v'-.8m'\z.\v'.5m'\z.\v'.5m'.\v'-.2m'
```

The zero width function can also be used to darken a character: \z*xx* will print an *x*, stay in place, and print it again.

The point-size change function is used frequently to make one character fit in proportion to another. The escape sequence for this function is \s. The change in size would be indicated by however many point sizes you want to change. \s-2 will cause a reduction of 2 point sizes while \s+2 will enlarge by 2 point sizes. For example, in making the "equals by definition" symbol ≜, the Δ was made slightly smaller to fit comfortably over the =. This was done by

```
\v'.3m'\z=\v'-.6m'\h'.3m'\s-1\(*D\s+1\v'.3m'
```

hence reducing the Δ (\(*D is the escape for this) by 1 point size before printing it, and then returning back to the previous size. \s0 can also be used to bring you back to the previous size.

Another handy tool is the font change function \f. Since EQN automatically sets its characters in italics, it will be necessary to specify any other font. To change fonts, use \f*x* where *x* is the desired font (R for roman, B for bold, I for italic, etc.). \fP will revert back to the previous font. A prime example of this would be the "angstrom" symbol Å:

```
\fR\zA\v'-.3m'\h'.2m'\(de\v'.3m'\fP\h'.2m'
```

where the capital "Å" is always printed in the Roman font.

A few words of caution. If in building your character you have used vertical motions or point-size or font changes, you must remember to undo them, or whatever follows will be off the main line or in the wrong size or font. It may also be necessary at times to use a horizontal motion to ensure enough space before and after your character. Also, you should test your character before putting it to actual use. Try it out with changes in original font or point size and see how it reacts to the change; don't be surprised if your character falls apart.

When you are satisfied with your character and it is ready to use, introduce it into your file with the define facility provided by EQN. For example, you would define the "not greater than" symbol (≯) as

```
define |>  % "\o'>\(or'" %
```

Now by using |> in an equation, you will get ≯.

In the previous example, the %'s and "'s are two different kinds of delimiters. The outside pair are essential to mark the beginning and end of the definition. (%'s were used, although any character will do as long as it is not used in the definition.) Also, any definition containing TROFF commands should be enclosed in " marks, so EQN will treat the TROFF commands as a unit.

4. Creating Special Characters for NEQN

To get a special character to print out on a computer terminal is not quite as involved as on the phototypesetter. You are restricted to working with only the characters available on the print wheel and movement is limited. Vertical motion can be obtained by \u and \d but this will only give you a motion of ½ line space per escape sequence. Spacing and backspacing is about all the horizontal motion you'll get. You can, however, use NEQN to define a character as was done with the "less than or approximately equal to" symbol ≲:

```
ndefine <wig % < from "~" %
```

All TROFF conventions work in NROFF, but because of the lack of characters, they may produce strange effects when using NEQN/NROFF. Therefore, it may be necessary to separately define characters for phototypesetter and computer terminal output. *tdefine* applies to definitions for EQN; *ndefine* works with NEQN. *define* uses the same definition for both.

Built-up characters for terminal output are usually just good enough for identification's sake; generally, they look lousy. However, given the time and additional effort, these characters can be refined so that their output on the terminal is quite satisfactory.

Acknowledgements

I would like to thank J. F. Ossanna, M. E. Lesk and other members of Center 127 for all their help; and special thanks to B. W. Kernighan for his teaching, guidance and patience throughout. I am especially grateful to S. P. Morgan for his encouragement and shared enthusiasm; without his aid and the concurrence of Dept. 7133K supervision, none of this would have been possible.

References

1. "A System for Typesetting Mathematics," B. W. Kernighan and L. L. Cherry, Computing Science Technical Report #17.
2. NROFF/TROFF User's Manual, J. F. Ossanna, BTL internal memorandum.

Appendix 1a

Additional Symbols from Phototypesetter Character Set

	character	EQN name
	¼	quarter
	¾	3quarter
	°	degree
	□	square
	○	circle
	■	blot
	●	bullet
	≅	–wig
	~	wig
	∝	prop
	∅	empty
	∈	member
	∪	cup
*	∩	cap
*	⊂	subset
*	⊃	supset
*	⊆	!subset
*	⊇	!supset

* NEQN/NROFF does not produce these symbols.

Appendix 1b

Additional Symbols Provided by EQN

	character	EQN name
*	½	half
	≅	approx
	⩾	>=
	⩽	<=
	≫	>>
	≪	<<
	→	->
	←	<-
	≡	==
	≠	!=
	±	+-
	∞	inf
	∂	partial
	′	prime
	·	cdot
	×	times
	∇	grad
	. . .	...
	, ... ,	,....,
	Σ	sum
	∫	int
	∏	prod
	∪	union
*	∩	inter

* NEQN/NROFF does not produce these symbols.

Appendix 2

Note: — represents a backspace in ndefined characters.

```
.EQ
tdefine ciplus % "\o'\(pl\(ci'" %
ndefine ciplus % O—+ %
tdefine citimes % "\o'\(mu\(ci'" %
ndefine citimes % O—x %
tdefine =wig % "\(eq\h'—\w'\(eq'u—\w'\s—2\(ap'u/2u'\v'—.4m'\s—2\z\(ap\(ap\s+2\v'.4m'\h'\w'\(eq'u—\w'\s—2\(ap'u/2u'" %
ndefine =wig % =—"" %
tdefine bigstar % "\o'\(pl\(mu'"%
ndefine bigstar % X—|—— %
tdefine =dot % "\z\(eq\v'—.6m'\h'.2m'\s+2.\s—2\v'.6m'\h'.1m'" %
ndefine =dot % = dot %
tdefine orsign % "\s—2\v'—.15m'\z\e\e\h'—.05m'\z\(sl\(sl\v'.15m'\s+2" %
ndefine orsign % \e/ %
tdefine andsign % "\s—2\v'—.15m'\z\(sl\(sl\h'—.05m'\z\e\e\v'.15m'\s+2" %
ndefine andsign % /\e %
tdefine =del % "\v'.3m'\z=\v'—.6m'\h'.3m'\s—1\("D\s+1\v'.3m'" %
ndefine =del % = to DELTA %
tdefine oppA % "\s—2\v'—.15m'\z\e\e\h'—.05m'\z\(sl\(sl\v'—.15m'\h'—.75m'\z—\z—\h'.2m'\z—\z—\v'.3m'\h'.4m'\s+2" %
ndefine oppA % V—— %
tdefine oppE %"\s—3\v'.2m'\z\(em\v'—.5m'\z\(em\v'—.5m'\z\(em\v'.55m'\h'.9m'\z\(br\z\(br\v'.25m'\s+3" %
ndefine oppE % E—/ %
tdefine incl % "\s—1\z\(or\h'—.1m'\v'—.45m'\z\(em\v'.7m'\z\(em\v'.2m'\(em\v'—.45m's+1" %
ndefine incl % C—_ %
tdefine nomem % "\o'\(mo\(sl'" %
ndefine nomem % C———/ %
tdefine angstrom % "\fR\zA\v'—.3m'\h'.2m'\(de\v'.3m'\fP\h'.2m'" %
ndefine angstrom % A to o %
tdefine star %{ roman "\v'.5m'\s+3"\s—3\v'—.5m'"}%
ndefine star % * %
tdefine || % \(or\(or %
tdefine <wig % "\z<\v'.4m'\(ap\v'—.4m'" %
ndefine <wig %{ < from "" }%
tdefine >wig % "\z>\v'.4m'\(ap\v'—.4m'" %
ndefine >wig %{ > from "" }%
tdefine langle % "\s—3\b'\(sl\e'\s0" %
ndefine langle %<%
tdefine rangle % "\s—3\b'\e\(sl'\s0" %
ndefine rangle %>%
tdefine hbar % "\zh\v'—.6m'\h'.05m'\(ru\v'.6m'" %
ndefine hbar % h—\u—\d %
ndefine ppd % _—| %
tdefine ppd % "\o'\(ru\s—2\(or\s+2'" %
tdefine <-> % "\o'\(<—\(—>'" %
ndefine <-> % "<——>" %
tdefine <=> % "\s—2\z<\v'.05m'\h'.2m'\z=\h'.55m'=\h'—.6m'\v'—.05m'>\s+2" %
ndefine <=> % "<=>" %
tdefine |< % "\o'<\(or'" %
ndefine |< % <—| %
tdefine |> % "\o'>\(or'" %
ndefine |> % |—> %
tdefine ang % "\v'—.15m'\z\s—2\(sl\s+2\v'.15m'\(ru" %
ndefine ang % /—_ %
tdefine rang % "\z\(or\h'.15m'\(ru" %
ndefine rang % L %
tdefine 3dot % "\v'—.8m'\z.\v'.5m'\z.\v'.5m'.\v'—.2m'" %
ndefine 3dot % .—\u.—\u.\d\d %
tdefine thf % ".\v'—.5m'.\v'.5m'." %
ndefine thf % ..—\u.\d %
.EN
```

The PWB/UNIX* document entitled:

PWB/UNIX View Graph and Slide Macros

is not yet available.

* UNIX is a Trademark/Service Mark of the Bell System.

BC — An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- to do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

BC — An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Bell Laboratories,
Murray Hill, New Jersey 07974

Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [6]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2,3]. Those who are familiar with C will find few surprises in this language.

Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators −, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7+−3
```

is interpreted to mean that −3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and −. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

a^b^c and a^(b^c)

are equivalent, as are the two expressions

a*b*c and (a*b)*c

BC shares with Fortran and C the undesirable convention that

a/b*c is equivalent to (a/b)*c

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

x = x + 3

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15 respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

Scaling

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
      auto z
      z = x*y
      return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

Control Statements

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
    auto a, b, c, d, n
    a = 1
    b = 1
    c = 1
    d = 0
    n = 1
    while(1==1){
        a = a*x
        b = b*n
        c = c + a/b
        n = n + 1
        if(c==d) return(c)
        d = c
    }
}
```

Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2,3] for their exact workings.

x=y=z is the same as	x=(y=z)
x =+ y	x = x+y
x =- y	x = x-y
x =* y	x = x*y
x =/ y	x = x/y
x =% y	x = x%y
x =^ y	x = x^y
x++	(x=x+1)-1
x--	(x=x-1)+1
++x	x = x+1
--x	x = x-1

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

Three Important Things

1. To exit a BC program, type 'quit'.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/*' and end with '*/'.

3. There is a library of math functions which may be obtained by typing at command level

```
bc -l
```

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [4].

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

Acknowledgement

The compiler is written in YACC [5]; its original version was written by S. C. Johnson.

References

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Fifth Edition (1974)

[2] D. M. Ritchie, *C Reference Manual.*

[3] B. W. Kernighan, *Programming in C: A Tutorial.*

[4] Robert Morris, *A Library of Reference Standard Mathematical Subroutines*, Internal memorandum, Bell Laboratories, 1975.

[5] S. C. Johnson, *YACC, Yet Another Compiler-Compiler.*

[6] R. Morris and L. L. Cherry, *DC — An Interactive Desk Calculator.*

Appendix

1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [] is optional.

2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

2.1. Comments

Comments are introduced by the characters /* and terminated by */.

2.2. Identifiers

There are three kinds of identifiers – ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named x, an array named x and a function named x, all of which are separate and distinct.

2.3. Keywords

The following are reserved keywords:

ibase	**if**
obase	**break**
scale	**define**
sqrt	**auto**
length	**return**
while	**quit**
for	

2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A–F are also recognized as digits with values 10–15, respectively.

3. Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

3.1. Primitive expressions

3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

3.1.1.2. *array-name*[*expression*]

Array elements are named expressions. They have an initial value of zero.

3.1.1.3. scale, ibase and obase

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

3.1.2. Function calls

3.1.2.1. *function-name*([*expression*[, *expression*...]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

3.1.2.2. sqrt(*expression*)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

3.1.2.3. length(*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

3.1.2.4. scale(*expression*)

The result is the scale of the expression. The scale of the result is zero.

3.1.3. Constants

Constants are primitive expressions.

3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

3.2. Unary operators

The unary operators bind right to left.

3.2.1. − *expression*

The result is the negative of the expression.

3.2.2. ++ *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

3.2.3. −− *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

3.2.4. *named-expression*++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

3.2.5. *named-expression*−−

The named expression is decremented by one. The result is the value of the named expression before decrementing.

3.3. Exponentiation operator

The exponentiation operator binds right to left.

3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is:

$\min(a \times b, \max(\text{scale}, a))$

3.4. Multiplicative operators

The operators *, /, % bind left to right.

3.4.1. *expression* * *expression*

The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is:

$\min(a+b, \max(\text{scale}, a, b))$

3.4.2. *expression / expression*

The result is the quotient of the two expressions. The scale of the result is the value of scale.

3.4.3. *expression % expression*

The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is *a-a/b*b*.*

The scale of the result is the sum of the scale of the divisor and the value of scale

3.5. Additive operators

The additive operators bind left to right.

3.5.1. *expression + expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

3.5.2. *expression - expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.6. assignment operators

The assignment operators bind right to left.

3.6.1. *named-expression = expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

3.6.2. *named-expression =+ expression*

3.6.3. *named-expression =- expression*

3.6.4. *named-expression =* expression*

3.6.5. *named-expression =/ expression*

3.6.6. *named-expression =% expression*

3.6.7. *named-expression =^ expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

4. Relations

Unlike all other operators, the relational operators are only valid as the object of an if, while, or inside a for statement.

4.1. *expression* < *expression*

4.2. *expression* > *expression*

4.3. *expression* <= *expression*

4.4. *expression* >= *expression*

4.5. *expression* == *expression*

4.6. *expression* != *expression*

5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

6.4. If statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

6.5. While statements

while(*relation***)** *statement*

The statement is executed while the relation is true The test occurs before each execution of the statement.

6.6. For statements

for(*expression*; *relation*; *expression***)** *statement*

The for statement is the same as
first-expression
while(*relation***)** {
    statement
    last-expression
}

All three expressions must be present.

6.7. Break statements

break

break causes termination of a **for** or **while** statement.

6.8. Auto statements

auto *identifier*[,*identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

6.9. Define statements

define([*parameter*[,*parameter*...]]**)**{
statements}

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

6.10. Return statements

return

return(*expression***)**

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

DC — An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

Bell Laboratories,
Murray Hill, New Jersey 07974

DC is an arbitrary precision arithmetic package implemented on the UNIX time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A—F which are treated as digits with values 10—15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ − * % ^

The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

s*x*

The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

l*x*

The value in register *x* is pushed onto the stack. The register *x* is not altered. If the l is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command l and is treated as an error by the command L.

d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[...]

puts the bracketed character string onto the top of the stack.

q

exits the program. If executing a string, the recursion level is popped by two. If q is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

<x >x =x !<x !>x !=x

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v

replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

!

interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If i is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If o is capitalized, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If k is capitalized, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

DETAILED DESCRIPTION

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0–99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always −1 and all other digits are in the range 0–99. The digit preceding the high order −1 digit is never a 99. The representation of −157 is 43,98,−1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,3

where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,−1 by the digit −1. In any case, digits which are not in the range 0−99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register scale and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity scale. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity scale and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with successive approximations by the rule

$$x_{n+1} = \tfrac{1}{2}(x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The hexadecimal digits A–F correspond to the numbers 10–15 regardless of input base. The i command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command I will push the value of the input base on the stack.

Output Commands

The command p causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command f. The o command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command O pushes the value of the output base on the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands s and l. The command s*x* pops the top of the stack and stores the result in register x. *x* can be any character. l*x* puts the contents of register x on the top of the stack. The l command has no effect on the contents of register *x*. The s command, however, is destructive.

Stack Commands

The command c clears the stack. The command d pushes a duplicate of the number on the top of the stack on the stack. The command z pushes the stack size on the stack. The command X replaces the number on the top of the stack with its scale factor. The command Z replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in [] pushes the ascii string on the stack. The q command quits or in executing a string, pops the recursion levels by two.

Internal Registers — Programming DC

The load and store commands together with [] to store strings, x to execute and the testing commands '<', '>', '=', '!<', '!>', '!=' can be used to program DC. The x command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lip1+ si li10>a]sa
0si lax
```

Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands S and L. S*x* pushes the top value of the main stack onto the stack for the register *x*. L*x* pops the stack for register *x* and puts the result on the main stack. The commands s and l also work on registers but not as push-down stacks. l doesn't effect the top of the register stack, and s destroys what was there before.

The commands to work on arrays are : and ;. :*x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. ;*x* is the command to load the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

Miscellaneous Commands

The command ! interprets the rest of the line as a UNIX command and passes it to UNIX to execute. One other compiler command is Q. This command uses the top of the stack as the number of levels of recursion to skip.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of scale were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for scale. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a scale to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

References

[1] L. L. Cherry, R. Morris, *BC — An Arbitrary Precision Desk-Calculator Language*,

[2] K. C. Knowlton, *A Fast Storage Allocator*, Comm. ACM 8, pp. 623-625 (Oct. 1965)

YACC — Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program which does input can be thought of as defining an "input language" which it accepts. The input languages may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, standard input facilities are restricted, difficult to use and change, and do not completely check their inputs for validity.

Yacc provides a general tool for controlling the input to a computer program. The Yacc user describes the structures of his input, together with code which is to be invoked when each such structure is recognized. Yacc turns such a specification into a subroutine which may be invoked to handle the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or, if he wishes, in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy specification.

Yacc is written in C[7], and runs under UNIX. The subroutine which is output may be in C or in Ratfor[4], at the user's choice; Ratfor permits translation of the output subroutine into portable Fortran[5]. The class of specifications accepted is a very general one, called LALR(1) grammars with disambiguating rules. The theory behind Yacc has been described elsewhere[1,2,3].

Yacc was originally designed to help produce the "front end" of compilers; in addition to this use, it has been successfully used in many application programs, including a phototypesetter language, a document retrieval system, a Fortran debugging system, and the Ratfor compiler.

YACC — Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

Section 0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules which describe the input structure, code which is to be invoked when these structures are recognized, and a low-level routine to do the basic input. Yacc then produces a subroutine to do the input procedure; this subroutine, called a *parser*, calls the user-supplied low-level input routine (called the *lexical analyzer)* to pick up the basic items (called *tokens)* from the input stream. These tokens are organized according to the input structure rules, called *grammar rules;* when one of these rules has been recognized, then the user code supplied for this rule, called an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma "," is quoted by single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

As we mentioned above, an important part of the input process is carried out by the lexical analyzer. This user routine reads the true input stream, recognizing those structures which are more conveniently or more efficiently recognized directly, and communicates these recognized tokens to the parser. For historical reasons, the name of a structure recognized by the lexical analyzer is called a *terminal symbol* name, while the name of a structure recognized by the parser is called a *nonterminal symbol* name. To avoid the obvious confusion of terminology, we shall usually refer to terminal symbol names as *token names.*

There is considerable leeway in deciding whether to recognize structures by the lexical analyzer or by a grammar rule. Thus, in the above example it would be possible to have other rules of the form

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    ...

month_name : 'D' 'e' 'c' ;
```

Here, the lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Rules of this sort tend to be a bit wasteful of time and space, and may even restrict the power of the input process (although they are easy to write). For a

more efficient input process, the lexical analyzer itself might recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters, such as ",", must also be passed through the lexical analyzer, and are considered tokens.

As an example of the flexibility of the grammar rule approach, we might add to the above specifications the rule

```
date : month '/' day '/' year ;
```

and thus optionally allow the form

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and a very small chance of disrupting existing input.

Frequently, the input being read does not conform to the specifications due to errors in the input. The parsers produced by Yacc have the very desirable property that they will detect these input errors at the earliest place at which this can be done with a left-to-right scan, thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling facilities, entered as part of the input specifications, frequently permit the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases probably represent true design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. The class of specifications which Yacc can handle compares very favorably with other systems of this type; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. In Section 4, we discuss the diagnostics produced when Yacc is unable to produce a parser from the given specifications. This section also describes a simple, frequently useful mechanism for handling operator precedences. Section 5 discusses error detection and recovery. Sections 6C and 6R discuss the operating environment and special features of the subroutines which Yacc produces in C and Ratfor, respectively. Section 7 gives some hints which may lead to better designed, more efficient, and clearer specifications. Finally, Section 8 has a brief summary. Appendix A has a brief example, and Appendix B tells how to run Yacc on the UNIX operating system. Appendix C has a brief description of mechanisms and syntax which are no longer actively supported, but which are provided for historical continuity with older versions of Yacc.

Section 1: Basic Specifications

As we noted above, names refer to either tokens or nonterminal symbols. Yacc requires those names which will be used as token names to be declared as such. In addition, for reasons which will be discussed in Section 3, it is usually desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent "%%" marks. (The per-cent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name or operator is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. Notice that the colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Notice that Yacc considers that upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'   represents newline
'\r'   represents return
'\''   represents single quote "'"
'\\'   represents backslash "\"
'\t'   represents tab
'\b'   represents backspace
'\xxx' represents "xxx" in octal
```

For a number of technical reasons, the nul character ('\0' or 000) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D  ;
A : E F  ;
A : G  ;
```

can be given to Yacc as

```
A :  B C D|
     E F|
     G ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easy to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :  ;
```

As we mentioned above, names which represent tokens must be declared as such. The simplest way of doing this is to write

```
%token  name1 name2 . . .
```

in the declarations section. (See Sections 3 and 4 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. If, by the end of the rules section, some nonterminal symbol has not appeared on the left of any rule, then an error message is produced and Yacc halts.

The left hand side of the *first* grammar rule in the grammar rules section has special importance; it is taken to be the controlling nonterminal symbol for the entire input process; in technical language it is called the *start symbol*. In effect, the parser is designed to recognize the start symbol; thus, this symbol generally represents the largest, most general structure described by the grammar rules.

The end of the input is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser subroutine returns to its caller when the endmarker is seen; we say that it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Frequently, the endmarker token represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

Section 2: Actions

To each grammar rule, the user may associate an action to be performed each time the rule is recognized in the input process. This action may return a value, and may obtain the values returned by previous actions in the grammar rule. In addition, the lexical analyzer can return values for tokens, if desired.

When invoking Yacc, the user specifies a programming language; currently, Ratfor and C are supported. An action is an arbitrary statement in this language, and as such can do input and output, call subprograms, and alter external vectors and variables (recall that a "statement" in both C and Ratfor can be compound and do many distinct tasks). An action is specified by an equal sign "=" at the end of a grammar rule, followed by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A: '(' B ')' = { hello( 1, "abc" ); }
```

and

```
XXX: YYY ZZZ =
    {
        printf("a message\n");
        flag = 25;
    }
```

are grammar rules with actions in C. A grammar rule with an action need not end with a semicolon; in fact, it is an error to have a semicolon before the equal sign.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some integer value. For example, an action which does nothing but return the value 1 is

```
= { $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the (integer) pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A: B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, we might have the rule

```
expression: '(' expression ')' ;
```

We wish the value returned by this rule to be the value of the expression in parentheses. Then we write

```
expression: '(' expression ')'    = { $$ = $2 ; }
```

As a default, the value of a rule is the value of the first element in it ($1). This is true even if there is no explicit action given for the rule. Thus, grammar rules of the form

```
A: B ;
```

frequently need not have an explict action.

Notice that, although the values of actions are integers, these integers may in fact contain pointers (in C) or indices into an array (in Ratfor); in this way, actions can return and reference more complex data structures.

Sometimes, we wish to get control before a rule is fully parsed, as well as at the end of the rule. There is no explicit mechanism in Yacc to allow this; the same effect can be obtained, however, by introducing a new symbol which matches the empty string, and inserting an action for this symbol. For example, we might have a rule describing an "if" statement:

```
statement: IF '(' expr ')' THEN statement
```

Suppose that we wish to get control after seeing the right parenthesis in order to output some code. We might accomplish this by the rules:

```
statement:  IF '(' expr ')' actn THEN statement
        = { call action1 }

actn:   /* matches the empty string */
        = { call action2 }
```

Thus, the new nonterminal symbol actn matches no input, but serves only to call action2 after the right parenthesis is seen.

Frequently, it is more natural in such cases to break the rule into parts where the action is needed. Thus, the above example might also have been written

```
statement:  ifpart THEN statement
        = { call action1 }

ifpart:     IF '(' expr ')'
        = { call action2 }
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines which build and maintain the tree structure desired. For example, suppose we have a C function "node", written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns a pointer to the newly created node. Then we can cause a parse tree to be built by supplying actions such as:

```
expr: expr '+' expr
        = { $$ = node( '+', $1, $3 ); }
```

in our specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in two places in the Yacc specification: in the declarations section, and at the head of the rules sections, before the first grammar rule. In each case, the declarations and definitions are enclosed in the marks "%{" and "%}". Declarations and definitions placed in the declarations section have global scope, and are thus known to the action statements and the lexical analyzer. Declarations and definitions placed at the head of the rules section have scope local to the action statements. Thus, in the above example, we might have included

```
%{ int variable 0; %}
```

in the declarations section, or, perhaps,

```
%{ static int variable; %}
```

at the head of the rules section. If we were writing Ratfor actions, we might want to include some COMMON statements at the beginning of the rules section, to allow for easy communication between the actions and other routines. For both C and Ratfor, Yacc has used only external names beginning in "yy"; the user should avoid such names.

Section 3: Lexical Analysis

The user must supply a lexical analyzer which reads the input stream and communicates tokens (with values, if desired) to the parser. The lexical analyzer is an integer valued function called yylex, in both C and Ratfor. The function returns an integer which represents the type of the token. The value to be associated in the parser with that token is assigned to the integer variable yylval. Thus, a lexical analyzer written in C should begin

```
yylex ( ) {
        extern int yylval;
        ...
```

while a lexical analyzer written in Ratfor should begin

```
integer function yylex(yylval)
      integer yylval
      ...
```

Clearly, the parser and the lexical analyzer must agree on the type numbers in order for communication between them to take place. These numbers may be chosen by Yacc, or chosen by the user. In either case, the "define" mechanisms of C and Ratfor are used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the specification. The relevant portion of the lexical analyzer (in C) might look like:

```
yylex( ) {
      extern int yylval;
      int c;
      ...
      c = getchar( );
      ...
      if( c >= '0' && c <= '9' ) {
            yylval = c-'0';
            return(DIGIT);
      }
      ...
```

The relevant portion of the Ratfor lexical analyzer might look like:

```
integer function yylex(yylval)
      integer yylval, digits(10), c
      ...
      data digits(1) / "0" /;
      data digits(2) / "1" /;
      ...
      data digits(10) / "9" /;
      ...
#     set c to the next input character
      ...
      do i = 1, 10 {
            if(c .EQ. digits(i)) {
                  yylval = i-1
                  yylex = DIGIT
                  return
            }
      }
      ...
```

In both cases, the intent is to return a token type of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification, the identifier DIGIT will be redefined to be equal to the type number associated with the token name DIGIT.

This mechanism leads to clear and easily modified lexical analyzers; the only pitfall is that it makes it important to avoid using any names in the grammar which are reserved or significant in the chosen language; thus, in both C and Ratfor, the use of token names of "if" or "yylex" will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name "error" is reserved for error handling, and should not be used naively (see Section 5).

As mentioned above, the type numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default type number for a literal character is the numerical value of the character, considered as a 1 byte integer. Other token names are assigned type numbers starting at 257. ! is a difficult, machine dependent operation to determine the numerical value of an input character in Ratfor (or Fortran). Thus, the Ratfor user of Yacc will probably wish to set his own type numbers, or not use any literals in his specification.

To assign a type number to a token (including literals), 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 type number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all type numbers be distinct.

There is one exception to this situation. For sticky historical reasons, the endmarker must have type number 0. Note that this is not unattractive in C, since the nul character is returned upon end of file; in Ratfor, it makes no sense. This type number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 as a type number upon reaching the end of their input.

Section 4: Ambiguity, Conflicts, and Precedence

A set of grammar rules is *ambiguous* if there is some input string which can be structured in two or more different ways. For example, the grammar rule

```
expr : expr '-' expr ;
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if we have input of the form

```
expr - expr - expr
```

the rule would permit us to treat this input either as

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(We speak of the first as *left association* of operators, and the second as *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input which it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. One valid thing for the parser to do is to *reduce* the input it has seen by applying this rule; after applying the rule, it would have reduced the input it had already seen to expr (the left side of the rule). It could then read the final part of the input:

```
- expr
```

and again reduce by the rule. We see that the effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr – expr

it could defer the immediate application of the rule, and continue reading (the technical term is *shifting*) the input until it had seen

expr – expr – expr

It could then apply the grammar rule to the rightmost three symbols, reducing them to expr and leaving

expr – expr

Now it can reduce by the rule again; the effect is to take the right associative interpretation. Thus, having read

expr – expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. We refer to this as a *shift/reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce/reduce conflict*.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule which describes which choice to make in a given situation is called a *disambiguating rule*.

Yacc has two disambiguating rules which are invoked by default, in the absence of any user directives to the contrary:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but the proper use of reduce/reduce conflicts is still a black art, and is properly considered an advanced topic.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. In these cases, the application of disambiguating rules is inappropriate, and leads to a parser which is in error. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts which were resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read, but there are no conflicts. For this reason, most previous systems like Yacc have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural to do, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat : IF '(' cond ')' stat |
       IF '(' cond ')' stat ELSE stat ;
```

Here, we consider IF and ELSE to be tokens, cond to be a nonterminal symbol describing conditional (logical) expressions, and stat to be a nonterminal symbol describing statements. In the following, we shall refer to these two rules as the *simple-if* rule and the *if-else* rule, respectively.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
    IF ( C2 ) S1
}
ELSE S2
```

or

```
IF ( C1 ) {
    IF ( C2 ) S1
    ELSE S2
}
```

The second interpretation is the one given in most programming languages which have this construct. Each ELSE is associated with the last preceding "un-ELSE'd" IF. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the ELSE. It can immediately *reduce* by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, we may *shift* the ELSE and read S2, and then reduce the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – we have a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

Notice that this shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the *state* of the parser, which is assigned a nonnegative integer. The number of states in the parser is typically two to five times the number of grammar rules.

When Yacc is invoked with the verbose (−v) option (see Appendix B), it produces a file of user output which includes a description of the states in the parser. For example, the output corresponding to the above example might be:

```
23: shift/reduce Conflict (Shift 45, Reduce 18) on ELSE

State 23

    stat : IF ( cond ) stat_
    stat : IF ( cond ) stat_ELSE stat

    ELSE    shift 45
    .       reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The state title follows, and a brief description of the grammar rules which are active in this state. The underline "_" describes the portions of the grammar rules which have been seen. Thus in the example, in state 23 we have seen input which corresponds to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The actions possible are, if the input symbol is ELSE, we may shift into state 45. In this state, we should find as part of the description a line of the form

```
stat : IF ( cond ) stat ELSE_stat
```

because in this state we will have read and shifted the ELSE. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not ELSE, we should reduce by grammar rule 18, which is presumably

```
stat : IF '(' cond ')' stat
```

Notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In most states, there will be only one reduce action possible in the state, and this will always be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here; in this case, a reference such as [1] might be consulted; the services of a local guru might also be appropriate.

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the area of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers which are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many

parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser which realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, which may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr :
        expr '=' expr |
        expr '+' expr |
        expr '-' expr |
        expr '*' expr |
        expr '/' expr |
        NAME ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. An interesting situation arises when we have a unary operator and a binary operator which have the same symbolic representation, but different precedences. An example is unary and binary '-'; frequently, unary minus is given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. We can indicate this situation by use of another keyword, %prec, to change the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal; it causes the precedence of the grammar rule to become that of the token name or literal. Thus, to make unary minus have the same precedence as multiplication, we might write:

```
%left '+' '-'
%left '*' '/'

%%

expr :
        expr '+' expr |
        expr '-' expr |
        expr '*' expr |
        expr '/' expr |
        '-' expr %prec '*' |
        NAME ;
```

Notice that the precedences which are described by %left, %right, and %nonassoc are independent of the declarations of token names by %token. A symbol can be declared by %token, and, later in the declarations section, be given a precedence and associativity by one of the above methods. It is true, however, that names which are given a precedence or associativity are also declared to be token names, and so in general do not need to be declared by %token, although it does not hurt to do so.

As we mentioned above, the precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals which have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Notice that some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule, or both, has no precedence and associativity associated with it, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

There are a number of points worth making about this use of disambiguation. There is no reporting of conflicts which are resolved by this mechanism, and these conflicts are not counted in the number of shift/reduce and reduce/reduce conflicts found in the grammar. This means that occasionally mistakes in the specification of precedences disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. Frequently, not enough operators or precedences have been specified; this leads to a number of messages about shift/reduce or reduce/reduce conflicts. The cure is usually to specify more precedences, or use the %prec mechanism, or both. It is generally good to examine the verbose output file to ensure that the conflicts which are being reported can be validly resolved by precedence.

Section 5: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid putting out any further output.

It is generally not acceptable to stop all processing when an error is found; we wish to continue scanning the input to find any further syntax errors. This leads to the problem of getting the parser "restarted" after an error. The general class of algorithms to do this involves reading ahead and discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser attempts to find the last time in the input when the special token "error" is permitted. The parser then behaves as though it saw the token name "error" as an input token, and attempts to parse according to the rule encountered. The token at which the error was detected remains the next input token after this error token is processed. If no special error rules have been specified, the processing effectively halts when an error is detected.

In order to prevent a cascade of error messages, the parser assumes that, after detecting an error, it remains in 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 error message is given, and the input token is quietly deleted.

As a common example, the user might include a rule of the form

```
statement : error ;
```

in his specification. This would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. (Notice, however, that it may be difficult or impossible to tell the end of a statement, depending on the other grammar rules). 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.

The user may supply actions after these special grammar rules, just as after the other grammar rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

The above form of grammar rule is very general, but somewhat difficult to control. Somewhat easier to deal with are rules of the form

```
statement : error ';' ;
```

Here, when there is an error, the parser will again attempt to skip over the statement, but in this case will do so by skipping to the next ";". All tokens after the error and before the next ";" give syntax errors, and are discarded. When the ";" is seen, this rule will be reduced, and any "cleanup" action associated with it will be performed.

Still another form of error rule arises in interactive applications, where we may wish to prompt the user who has incorrectly input a line, and allow him to reenter the line. In C we might write:

```
inputline: error '\n' prompt inputline
     = { $$ = $4; };

prompt:   /* matches no input */
     = { printf( "Reenter last line: " ); };
```

There is one difficulty with this approach; the parser must correctly process three input tokens before it is prepared to admit that it has correctly resynchronized after the error. Thus, if the reentered line contains errors in the first two tokens, the parser will simply delete the offending tokens, and give no message; this is clearly unacceptable. For this reason, there is a mechanism in both C and Ratfor which can be used to force the parser to believe that resynchronization has taken place. One need only include a statement of the form

```
yyerrok ;
```

in his action after such a grammar rule, and the desired effect will take place; this name will be expanded, using the "# define" mechanism of C or the "define" mechanism of Ratfor, into an appropriate code sequence. For example, in the situation discussed above where we want to prompt the user to produce input, we probably want to consider that the original error has been recovered when we have thrown away the previous line, including the newline. In this case, we can reset the error state before putting out the prompt message. The grammar rule for the nonterminal symbol prompt becomes:

```
prompt:   /* matches no input */
     = {
          yyerrok;
          printf( "Reenter last line: " );
     } ;
```

There is another special feature which the user may wish to use in error recovery. As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is seen to be 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 user wishes a way of clearing the previous input token held in the parser. One need only include a statement of the form

```
yyclearin ;
```

in his action; again, this expands, in both C and Ratfor, to the appropriate code sequence. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, which attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; we wish to throw away the old, illegal token, and reset the error state. We might do this by the sequence:

```
statement : error
     = {
          resynch( );
          yyerrok ;
          yyclearin ;
     } ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors, and have the virtue that the user can get "handles" by which he can deal with the error actions required by the lexical and output portions of the system.

Section 6C: The C Language Yacc Environment

The default mode of operation in Yacc is to write actions and the lexical analyzer in C. This has a number of advantages; primarily, it is easier to write character handling routines, such as the lexical analyzer, in a language which supports character-by-character I/O, and has shifting and masking operators.

When the user inputs a specification to Yacc, the output is a file of C programs, called "y.tab.c". These are then compiled, and loaded with a library; the library has default versions of a number of useful routines. This section discusses these routines, and how the user can write his own routines if desired. The name of the Yacc library is system dependent; see Appendix B.

The subroutine produced by Yacc is called "yyparse"; it is an integer valued function. When it is called, it in turn repeatedly calls "yylex", the lexical analyzer supplied by the user (see Section 3), to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse returns the value 1, or the lexical analyzer returns the endmarker token (type number 0), and the parser accepts. In this case, yyparse returns the value 0.

Three of the routines on the Yacc library are concerned with the "external" environment of yyparse. There is a default "main" program, a default "initialization" routine, and a default "accept" routine, respectively. They are so simple that they will be given here in their entirety:

```
main( argc, argv )
int argc;
char *argv[ ]
{
      yyinit( argc, argv );
      if( yyparse( ) )
            return;
      yyaccpt( );
}

yyinit( ) { }

yyaccpt( ) { }
```

By supplying his own versions of yyinit and/or yyaccpt, the user can get control either before the parser is called (to set options, open input files, etc.) or after the accept action has been done (to close files, call the next pass of the compiler, etc.). Note that yyinit is called with the two "command line" arguments which have been passed into the main program. If neither of these routines is redefined, the default situation simply looks like a call to the parser, followed by the termination of the program. Of course, in many cases the user will wish to supply his own main program; for example, this is necessary if the parser is to be called more than once.

The other major routine on the library is called "yyerror"; its main purpose is to write out a message when a syntax error is detected. It has a number of hooks and handles which attempt to make this error message general and easy to understand. This routine is somewhat more complex, but still approachable:

```
extern int yyline;  /* input line number */

yyerror(s)
char *s;
{
      extern int yychar;
      extern char *yysterm[ ];

      printf("\n%s", s );
      if( yyline )
            printf(", line %d,", yyline );
      printf(" on input: ");
      if( yychar >= 0400 )
            printf("%s\n", yysterm[yychar-0400] );
      else switch ( yychar ) {
      case '\t': printf( "\\t\n" ); return;
      case '\n': printf( "\\n\n" ); return;
      case '\0': printf( "$end\n" ); return;
      default: printf( "%c\n" , yychar ); return;
      }
}
```

The argument to yyerror is a string containing an error message; most usually, it is "syntax error". yyerror also uses the external variables yyline, yychar, and yysterm. yyline is a line number which, if set by the user to a nonzero number, will be printed out as part of the error message. yychar is a variable which contains the type number of the current token. yysterm has the names, supplied by the user, for all the tokens which have names. Thus, the routine spends most of its time trying to print out a reasonable name for the input token. The biggest problem with the routine as given is that, on Unix, the error message does not go out on the error file (file 2). This is hard to arrange in such a way that it works with both the portable I/O library and the system I/O library; if a way can be worked out, the routine will be changed to do this. *Beware:* This routine will not work if any token names have been given redefined type numbers. In this case, the user must supply his own yyerror routine. Hopefully, this "feature" will disappear soon.

Finally, there is another feature which the C user of Yacc might wish to use. The integer variable yydebug is normally set to 0. If it is set to 1, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

Section 6R: The Ratfor Language Yacc Environment

For reasons of portability or compatibility with existing software, it may be desired to use Yacc to generate parsers in Ratfor, or, by extension, in portable Fortran. The user is likely to work considerably harder doing this than he might if he were to use C.

When the user inputs a specification to Yacc, and specifies the Ratfor option (see Appendix B), the output is a file of Ratfor programs called "y.tab.r". These programs are then compiled, and provide the desired subroutine.

The subroutine produced by Yacc which does the input process is an integer function called "yypars". When it is called, it in turn repeatedly calls "yylex", the lexical analyzer supplied by the user (see Section 3). Eventually, either an error is detected, in which case (if no error recovery is possible) yypars returns the value 1, or the lexical analyzer returns the endmarker (type number 0), and the parser accepts. In this case, yypars returns 0.

Unlike the C program situation (see Section 6C) there is no library of Ratfor routines which must be used in the loading process. As a side effect of this, *the user must supply a main program which calls yypars*. A suggested Ratfor main program is

```
integer yypars
n = yypars(0)
if( n .EQ. 0 ) {
        ... here if the program accepted
} else {
        ... here if there were unrecoverable errors
}
end
```

Notice that there is no easy way for the user to get control when an error is detected, since the Fortran language provides only a very crude character string capability.

There is another feature which the Ratfor user might wish to use. The argument to yypars is normally 0. If it is set to 1, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. During the input process, the value of this debug flag is kept in a common variable yydebu, which is available to the actions and may be set and reset at will.

Statement labels 1 through 1000 are reserved for the parser, and may not appear in actions; note that, because Ratfor has a more modern control structure than Fortran, it is rarely necessary to use statement labels at all; the most frequent use of labels in Ratfor is in formatted I/O.

Because Fortran has no standard character set and not even a standard character width, it is difficult to produce a lexical analyzer in portable Fortran The usual solution is to provide a routine which does a table search to get the internal type number for each input character, with the understanding that such a routine can be recoded to run far faster for any particular machine.

Finally, we must warn the user that the Ratfor feature of Yacc has been operational for a much shorter time than the other portions of the system. If past experience is any guide, the Ratfor support will develop and become more powerful and better human engineered in response to user complaints and requirements. Thus, the potential Ratfor user might do well to contact the author to discuss his own particular needs.

Section 7. Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are, more or less, independent; the reader seeing Yacc for the first time may well find that this entire section could be omitted.

Input Style

It is difficult to input rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan, and are officially endorsed by the author.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Indent rule bodies by one tab stop, and action bodies by two tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Common Actions

When several grammar rules have the same action, the user might well wish to provide only one code sequence. A simple, general mechanism is, of course, to use subroutine calls. It is also possible to put a label on the first statement of an action, and let other actions be simply a goto to this label. Thus, if the user had a routine which built trees, he might wish to have only one call to it, as follows:

```
expr :
        expr '+' expr =
        { binary:
                $$ = btree( $1, $2, $3 );
        } |
        expr '-' expr =
        {
                goto binary;
        } |
        expr '*' expr =
        {
                goto binary;
        } ;
```

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list :
        item |
        list ',' item ;
```

and

```
sequence :
        item |
        sequence item ;
```

Notice that, in each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

If the user were to write these rules right recursively, such as

```
sequence :
        item |
        item sequence ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

The user should also consider whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
sequence :
        | /* empty */
        sequence item ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Experience suggests that permitting empty sequences leads to increased generality, which frequently is not evident at the time the rule is first written. There are cases, however, when the Yacc algorithm can fail when such a change is made. In effect, conflicts might arise when Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know! Nevertheless, this principle is still worth following wherever possible.

Lexical Tie-ins

Frequently, there are lexical decisions which depend on the presence of various constructions in the specification. 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 these situations is to create a global flag which is examined by the lexical analyzer, and set by actions. For example, consider a situation where we have a program which consists of 0 or more declarations, followed by 0 or more statements. We declare a flag called "dflag", which is 1 during declarations, and 0 during statements. We may do this as follows:

```
%{
        int dflag ;
%}
%%
program :
        decls  stats ;

decls :
        = /* empty */
        {
                dflag = 1;
        } |
        decls declaration ;

stats :
        = /* empty */
        {
                dflag = 0;
        } |
        stats statement ;

        . . . other rules . . .
```

The flag dflag is now set to zero when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can

tell that the declaration section has ended and the statements have begun. Frequently, however, this single token exception does not affect the lexical scan required.

Clearly, this kind of "backdoor" approach can be elaborated on to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Bundling

Bundling is a technique for collecting together various character strings so that they can be output at some later time. It is derived from a feature of the same name in the compiler/compiler TMG [6].

Bundling has two components – a nice user interface, and a clever implementation trick. They will be discussed in that order.

The user interface consists of two routines, "bundle" and "bprint".

```
bundle( a1, a2, . . ., an )
```

accepts a variable number of arguments which are either character strings or bundles, and returns a bundle, whose value will be the concatenation of the values of a1, . . ., an.

```
bprint( b )
```

accepts a bundle as argument and outputs its value.

For example, suppose that we wish to read arithmetic expressions, and output function calls to routines called "add", "sub", "mul", "div", and "assign". Thus, we wish to translate

```
a = b – c*d
```

into

```
assign(a,sub(b,mul(c,d)))
```

A Yacc specification file which does this is given in Appendix D; this includes an implementation of the bundle and bprint routines. A rule and action of the form

```
expr:
        expr '+' expr =
        {
                $$ = bundle( "add(", $1, ",", $3, ")" );
        }
```

causes the returned value of expr to be come a bundle, whose value is the character string containing the desired function call. Each NAME token has a value which is a pointer to the actual name which has been read. Finally, when the entire input line has been read and the value has been bundled, the value is written out and the bundles and names are cleared, in preparation for the next input line.

Bundles are implemented as arrays of pointers, terminated by a zero pointer. Each pointer either points to a bundle or to a character string. There is an array, called *bundle space*, which contains all the bundles.

The implementation trick is to check the values of the pointers in bundles – if the pointer points into bundle space, it is assumed to point to a bundle; otherwise it is assumed to point to a character string.

The treatment of functions with a variable number of arguments, like bundle, is likely to differ from one implementation of C to another.

In general, one may wish to have a simple storage allocator which allocates and frees bundles, in order to handle situations where it is not appropriate to completely clear all of bundle space at one time.

Reserved Words

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, since it is difficult to pass the required information to the lexical analyzer which tells it "this instance of if is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement, and one will probably be supported eventually. Until this day comes, I suggest that the keywords be *reserved;* that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway (he said weakly . . .).

Non-integer Values

Frequently, the user wishes to have values which are bigger than integers; again, this is an area where Yacc does not make the job as easy as it might, and some additional support is likely. Nevertheless, at the cost of writing a storage manager, the user can return pointers or indices to blocks of storage big enough to contain the full values desired.

Previous Work

There have been many previous applications of Yacc. The user who is contemplating a big application might well find that others have developed relevant techniques, or even portions of grammars. Yacc specifications appear to be easier to change than the equivalent computer programs, so that the "prior art" is more relevant here as well.

Section 8: User Experience, Summary, and Acknowledgements

Yacc has been used in the construction of a C compiler for the Honeywell 6000, a system for typesetting mathematical equations, a low level implementation language for the PDP 11, APL and Basic compilers to run under the UNIX system, and a number of other applications.

To summarize, Yacc can be used to construct parsers; these parsers can interact in a fairly flexible way with the lexical analysis and output phases of a larger system. The system also provides an indication of ambiguities in the specification, and allows disambiguating rules to be supplied to resolve these ambiguities.

Because the output of Yacc is largely tables, the system is relatively language independent. In the presence of reasonable applications, Yacc could be modified or adapted to produce subroutines for other machines and languages. In addition, we continue to seek better algorithms to improve the lexical analysis and code generation phases of compilers produced using Yacc.

This document would be incomplete if I did not give credit 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. Al Aho also deserves recognition for bringing the mountain to Mohammed, and other favors.

References

1 Aho, A.V. and Johnson, S.C., "LR Parsing", Computing Surveys, Vol 6, No 2, June 1974, pp. 99-124.

2 Aho, A.V., Johnson, S.C., and Ullman, J.D., "Deterministic Parsing of Ambiguous Grammars", Proceedings of the A.C.M. Symposium on Principles of Programming Languages, October 1973, pp. 1-21; to appear in CACM.

3 Aho, A.V. and Ullman, J.D., Theory of Parsing, Translation, and Compiling. Volume 1 (1972) and Volume 2 (1973), Prentice-Hall, Englewood Cliffs, N.J.

4 Kernighan, B. W., Ratfor, a Rational Fortran

5 Ryder, B. B., "The PFORT Verifier," Software—Practice and Experience, Vol 4 (1974), pp 359-377.

6 McIlroy, M. D., A Manual for the TMG Compiler-writing Language

7 Ritchie, D. M., C Reference Manual

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled a through z, and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression is an assignment at the top level, the value is not printed; otherwise it is. As in C, an integer which begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing the way that precedences and ambiguities are used, as well as showing how simple error recovery operates. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; frequently, this job is better done by the lexical analyzer.

```
%token DIGIT LETTER     /* these are token names */
%left '|'               /* declarations of operator precedences */
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS            /* supplies precedence for unary minus */
%{                      /* declarations used by the actions */
     int base;
     int regs[26];
%}

%%  /* beginning of rules section */

list :      /* list is the start symbol */
     |      /* empty */
     list stat '\n' |
     list error '\n' =
     {
          yyerrok ;
     } ;

stat :
     expr =
     {
          printf("%d\n", $1) ;
     } |
     LETTER '=' expr =
     {
          regs[$1] = $3 ;
     } ;

expr :
     '(' expr ')' =
     {
          $$ = $2 ;
     } |
```

```
        expr '+' expr =
        {
                $$ = $1 + $3 ;
        }|
        expr '-' expr =
        {
                $$ = $1 - $3 ;
        }|
        expr '*' expr =
        {
                $$ = $1 * $3 ;
        }|
        expr '/' expr =
        {
                $$ = $1 / $3 ;
        }|
        expr '%' expr =
        {
                $$ = $1 % $3 ;
        }|
        expr '&' expr
        {
                $$ = $1 & $3 ;
        }|
        expr '|' expr
        {
                $$ = $1 | $3 ;
        }|
        '-' expr  %prec UMINUS
        {
                $$ = - $2 ;
        }|
        LETTER
        {
                $$ = regs[$1] ;
        }|
        number ;

number :
        DIGIT =
        {
                $$ = $1 ;
                base = 10 ;
                if( $1 == 0 )
                        base = 8 ;
        }|
        number DIGIT =
        {
                $$ = base * $1 + $2 ;
        } ;

%%      /* start of programs */
```

```
yylex( ) /* lexical analysis routine */
{
      /* returns LETTER for a lower case letter, yylval = 0 through 25 */
      /* return DIGIT for a digit, yylval = 0 thro' gh 9 */
      /* all other characters are returned immediately */

      int c ;

      while( (c=getchar( )) == ' ' )
            ;
      if( c >= 'a' && c <= 'z' ) {
            yylval = c - 'a' ;
            return( LETTER ) ;
      }
      if( c >= '0' && c <= '9' ) {
            yylval = c - '0' ;
            return( DIGIT ) ;
      }
      return( c ) ;
}
```

Appendix B: Use of Yacc on Unix

Suppose that the Yacc specification is on a file called yfile. If the actions are in C, Yacc is invoked by

```
yacc yfile
```

The output appears on file y.tab.c To compile the parser and load it with the Yacc library, use the command

```
cc y.tab.c -ly
```

If Yacc is invoked with the option −v:

```
yacc −v yfile
```

a verbose description of the parser is produced on file y.output. The C user should consult section 6C for more information about the run time environment.

If the actions are in Ratfor, the user should invoke Yacc with the option −r:

```
yacc −r yfile
```

The Ratfor output appears on file y.tab.r It may be compiled by

```
rc −2 y.tab.r
```

Note that when Yacc is used to produce Ratfor programs, there is no need to load these programs with any library.

If the −v action is also invoked:

```
yacc −rv yfile
```

a verbose description of the parser is produced on file y.output. The Ratfor user should consult section 6R for more information about the run time environment.

Appendix C: Old Features Supported but not Encouraged

This appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may be delimited by double quotes "" as well as single quotes ''.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. The curly braces "{" and "}" around an action are optional if the action consists of a single C statement. (They are always required in Ratfor).

Appendix D: An Example of Bundling

The following program is an example of the technique of bundling; this example is discussed in Section 7.

```
/* warnings:
1. This works on Unix; the handling of functions with a variable number of arguments is
   different on different systems.
2. A number of checks for array bounds have been left out to avoid obscuring the basic
   ideas, but should be there in a practical program.
   */

%token NAME

%right '='
%left '+' '-'
%left '*' '/'

%%

lines :
        = /* empty */
        {
                bclear( ) ;
        } |
        lines expr '\n' =
        {
                bprint( $2 ) ;
                printf( "\n" ) ;
                bclear( ) ;
        } |
        lines error '\n' =
        {
                bclear( ) ;
                yyerrok;
        } ;

expr :
        expr '+' expr =
        {
                $$ = bundle( "add(", $1, ",", $3, ")" );
        } |
        expr '-' expr =
        {
                $$ = bundle( "sub(", $1, ",", $3, ")" );
        } |
        expr '*' expr =
        {
                $$ = bundle( "mul(", $1, ",", $3, ")" );
        } |
```

```
        expr '/' expr
        {
                $$ = bundle( "div(", $1, ",", $3, ")" );
        }|
        '(' expr ')' =
        {
                $$ = $2;
        }|
        NAME '=' expr =
                $$ = bundle( "assign(", $1, ",", $3, ")" );
        }|
        NAME ;

%%

#define     nsize 200
char  names[nsize], *nptr { names };

#define     bsize 500
int   bspace[bsize], *bptr { bspace };

yylex( )
{
        int c;

        c = getchar( );
        while( c == ' ' )
                c = getchar( );
        if( c>='a' && c<='z' ) {
                yylval = nptr;
                for( ; c>='a' && c<='z'; c=getchar( ) )
                        *nptr++ = c;
                ungetc( c );
                *nptr++ = '\0';
                return( NAME );
        }
        return( c );
}

bclear( )
{


        nptr = names;
        bptr = bspace;
}

bundle( a1,a2,a3,a4,a5 )
{
        int i, j, *p, *obp;

        p = &a1;
        i = nargs( );
```

```
        obp = bptr;

        for( j=0; j<i; ++j )
                *bptr++ = *p++;
        *bptr++ = 0;
        return( obp );
}

bprint( p )
int *p;
{

        if( p>=bspace && p< &bspace[bsize] ) /* bundle */
                while( *p != 0 )
                        bprint( *p++ );
        else printf( "%s", p );
}
```

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt
Bell Laboratories
Murray Hill, New Jersey 07974

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.

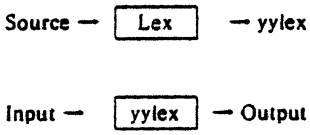
Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Table of Contents

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



An overview of Lex

Figure 1

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 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 there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

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]+$    ;
```

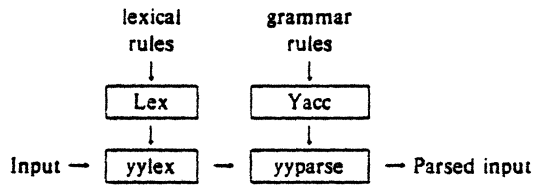
is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior 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.

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 [3]. 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 [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time



Lex with Yacc

Figure 2

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 (in C) or branches of a computed GOTO (in Ratfor). 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 *cd*. . . 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 [5]. 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. The usual C escapes with \ are recognized: \n is newline, \t is tab, \r is return, 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 *[ab]* 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, b, or 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]
```

in C or

```
yytext(yyleng)
```

in Ratfor.

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 *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, 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. There is another important routine in Ratfor, named *lexshf,* which is described below under "Character Set". 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 $ 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 containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

In Ratfor all of the standard I/O library routines, *input,*

output, ***unput***, ***yywrap***, and ***lexshf***, are defined as integer functions. This requires ***input*** and ***yywrap*** to be called with arguments. One dummy argument is supplied and ignored.

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 begining 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                   [TEde][-+]?{D}+
%%
{D}+                printf("integer");
{D}+"."{D}*({E})?   |
{D}*"."{D}+({E})?   |
{D}+{E}
```

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* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file *%R*.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file *%C*.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll*. So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

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. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c-./lexclib h=	./rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(cload)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text*. The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

a. Edit the Ratfor program.
1. Remove all tabs.
2. Change all lower case letters to upper case letters.
3. Convert the file to an 80-column card image file.
b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.
c. Compile the Fortran.
d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. *Warning:* Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

8 Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program 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 -lS
```

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]+      {
            scanf(-1, yytext, "%d", &k);
            if (k%7 == 0)
                printf("%d", k+3);
            else
                printf("%d",k);
            }
```

to do just that. The rule [0-9]+ recognizes strings of digits; *scanf* 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]+                {
                        scanf(-1, yytext, "%d", &k);
                        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 than 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}l{m}{a}{c}{h}    {yytext[0] =+ 'r' - 'd';
```

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. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine *lexshf* is called to change the representation delivered by *input* into a right-adjusted integer. If the user changes the I/O library, the routine *lexshf* should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter *a* as in the Fortran value *1Ha* while in C the letter *a* is represented 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 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

```
%T
1     Aa
2     Bb
...
26    Zz
27    \n
28    +
29    -
30    0
31    1
...
39    9
%T
```

Sample character table.

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.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace *input, output, unput* and *lexshf* but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave *input* and *output* as routines that read with 80A1 format, but replace *lexshf* by a table lookup routine.

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) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.

7) 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.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context. And any significant Lex source is too big for the IBM C compiler when translated.

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. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C Programming Language*, Computing Science Technical Report No. 31 (1975). Bell Laboratories, Murray Hill, NJ 07974.
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran*, to appear in *Software Practice and Experience, 1975.*
3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Comm. ACM **18**, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor*, Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library*, contained in reference [1], above.

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

- statement grouping
- **if-else** and **switch** for decision-making
- **while**, **for**, **do**, and **repeat-until** for looping
- **break** and **next** for controlling loop exits

and some "syntactic sugar":

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of >, >=, etc., into .GT., .GE., etc.
- **return**(expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other env ironments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least a dozen different types of computers at over one hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

January 1, 1977

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "I to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N−1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran.* Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation: the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like "x>100"), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of H's: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a transla-

tion. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to -1 if **x** is less than zero, to $+1$ if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        - - -
else if (...)
        - - -
else if (...)
        - - -
...
else
        - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-else'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {

        case expr1 :
                statements
        case expr2, expr3 :
                statements
        ...
        default:
                statements
}
```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression*, and there is a **default** section, the statements with it are done; if there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10      continue
```

The syntax is:

```
do legal-Fortran-DO-text
        Ratfor statement
```

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do**.

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the

do; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with I set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k {
                ---
        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
    # returns sin(x) to accuracy e, by
    # sin(x) = x - x**3/3! + x**5/5! - ...

    sin = x
    term = x

    i = 3
    while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
    }

    return
    end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x****3 and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal Fortran condition)
        Ratfor statement
```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while**. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
}
```

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
        Ratfor statement
```

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init*, *condition*, and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) != blank)
                break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
          I = 81 - J
          IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
        Ratfor statement
until (legal Fortran condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value −1.

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1) {
                        equal = 1
                        return
                }
        equal = 0
        return
        end
```

In many languages (e.g., PL/1) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function **F**, **return**(expression) is equivalent to

```
{ F = expression; return }
```

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1)
                        return(1)
        return(0)
        end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
=   +   -   *   ,   |   &   (   _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
        write(6, 100)
100     format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	¬	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	(	]	)
$(	(	$)	)

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)

        if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define    YES     1
define    NO      0
define    EOS     -1
define    ARB     100

# equal _ compare str1 to str2;
#     return YES if equal, NO if not
      integer function equal(str1, str2)
      integer str1(ARB), str2(ARB)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == EOS)
                  return(YES)
      return(NO)
      end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
      include commonblocks
      ...
      end

suroutine y
      include commonblocks
      ...
      end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog   : stat
       | prog  stat
stat   : if (...) stat
       | if (...) stat else stat
       | while (...) stat
       | for (...; ...; ...) stat
       | do ... stat
       | repeat stat
       | repeat stat until (...)
       | switch (...) { case ...: prog ...
                        default: prog }
       | return
       | break
       | next
       | digits  stat
       | { prog }
       | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the **else**. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve

distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v \pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain

that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience*, October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter*, August 1975.

[4] D. M. Ritchie, B. W. Kernighan and M. E. Lesk, "The C Programming Language," Bell Laboratories Computing Science Technical Report #31, 1975.

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1974.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

Appendix: Usage on UNIX and GCOS.

Beware — local customs vary. Check with a native before going into the jungle.

UNIX

The program **ratfor** is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **−6x**, which uses **x** as a continuation character in column 6 (UNIX uses **&** in column 1), and **−C**, which causes Ratfor comments to be copied into the generated Fortran.

The program **rc** provides an interface to the **ratfor** command which is much the same as **cc**. Thus

rc [options] files

compiles the files specified by **files**. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags **−C** and **−6x** described above are recognized, as are

−c	compile only; don't load
−f	save intermediate Fortran .f files
−r	Ratfor only; implies −c and −f
−2	use big Fortran compiler (for large programs)
−U	flag undeclared variables (not universally available)

Other flags are passed on to the loader.

GCOS

The program **./ratfor** is the bare translator, and is identical to the UNIX version, except that the continuation convention is **&** in column 6. Thus

./ratfor files >output

translates the Ratfor source on **files** and collects the generated Fortran on file 'output' for subsequent processing.

./rc provides much the same services as **rc** (within the limitations of GCOS), regrettably with a somewhat different syntax. Options recognized by **./rc** include

name	Ratfor source or library, depending on type
h=/name	make TSS H• file (runnable version); run as /name
r=/name	update and use random library
a=	compile as ascii (default is bcd)
C=	copy comments into Fortran
f=name	Fortran source file
g=name	gmap source file

Other options are as specified for the **./cc** command described in [4].

TSO, TSS, and other systems

Ratfor exists on various other systems; check with the author for specifics.

A.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

M4 is a macro processor available on UNIX and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

April 1, 1977

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is '−', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines N to be 100, and uses this "symbolic constant" in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name

is not followed immediately by '(', it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains a lot of N's.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if N is redefined? Or, to say it another way, is M defined as N or as 100? In M4, the latter is true — M is 100, so even if N subsequently changes, M does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string N is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when you ask for M later, you'll always get the value of N at that time (because the M will be replaced by N which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string N, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining N:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the N in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine N, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ' are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

changequote

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of N. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **gcos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix', `define(wordsize,16)' )
ifdef(`gcos', `define(wordsize,36)' )
```

makes a definition appropriate for the particular machine. Don't forget the quotes!

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(`unix', on UNIX, not on UNIX)
```

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the nth argument when the macro is actually used. Thus, the macro **bump**, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$4 through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

```
define(a,  b  c)
```

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, `incr(N)')
```

Then N1 is defined as one more than the current value of N.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

```
unary + and -
** or ^        (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
!             (not)
& or &&       (logical and)
| or ||       (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want M to be 2**N+1. Then

```
define(N, 3)
define(M, `eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

```
include(filename)
```

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

define(compare, `ifelse($1, $2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr(`now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or −1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

translit(s, aeiou)

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
    define(...)
    ...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

errprint(`fatal error')

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

3 changequote(L, R)
1 define(name, replacement)
4 divert(number)
4 divnum
5 dnl
5 dumpdef(`name', `name', ...)
5 errprint(s, s, ...)
4 eval(numeric expression)
3 ifdef(`name', this if true, this if false)
5 ifelse(a, b, c, d)
4 include(file)
3 incr(number)
5 index(s1, s2)
5 len(string)
4 maketemp(...XXXXX...)
4 sinclude(file)
5 substr(string, position, number)
4 syscmd(s)
5 translit(str, from, to)
3 undefine(`name')
4 undivert(number,number,...)

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Inc., 1976.

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

```
think — edit — make — test . . .
```

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *IS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
       cc x.o y.o z.o  -lS  -o prog

x.o y.o :  defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "cc −c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make's* innate knowledge:

```
prog : x.o y.o z.o
       cc x.o y.o z.o  -lS  -o prog
x.o : x.c defs
       cc  -c x.c
y.o : y.c defs
       cc  -c y.c
z.o : z.c
       cc  -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c)* would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make's* ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command

strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
  . . .
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make "LIBES= -ll -lp"
```

loads them with both the Lex ("-ll") and the portable ("-lp") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in Unix commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as in the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lp
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]
[*(tab)* commands] [# . . .]
. . .

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "−i" flags has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some Unix commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *chdir* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

−p Print out the complete set of macro definitions and target descriptions

−d Debug mode. Print out detailed information on files and times examined.

−f Description file name. The next argument is assumed to be the name of a description file. A file name of "−" denotes the standard input. If there are no "−f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

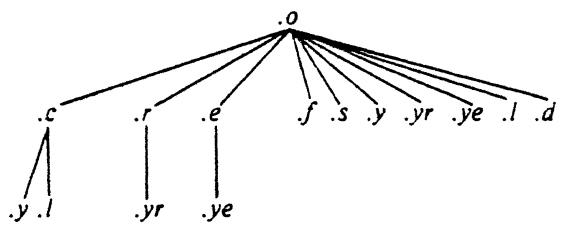
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before

compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS= -O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command
P = und -3 | opr -r2        # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O
make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make
$(OBJECTS): defs
gram.o: lex.c
cleanup:
        -rm *.o gram.c
        -du
install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make
print: $(FILES)     # print recently changed files
        pr $? | $P
        touch print
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap
lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from

typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
        or
make print "P= cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make's* specific meaning of dependency. If file *x.c* has a "#include "defs" " line, then the object file *x.o* depends on *defs;* the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o.)*

To discover what *make* would do, the "-n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "-t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("-d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. J. Johnson, "Yacc — Yet Another Compiler-Compiler", *Computing Science Technical Report #32*, July 1975.
2. M. E. Lesk, "Lex — A Lexical Analyzer Generator", *Computing Science Technical Report #39*, October 1975.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "−r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ".*r*" file to a ".*o*" file is thus ".*r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file, the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
        $(AS) -o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```