ConvexOS Tutorial Papers

Eighth Edition

CONVEX COMPUTER CORPORATION

606

CONVEX Computer Corporation
P.O. Box 833851
Richardson, TX 75083-3851
(214) 497-4000

CONVEX
Tutorial Papers

Order No. DSW-002

Eighth Edition
November 1991

CONVEX Press
Richardson, Texas USA

CONVEX
Tutorial Papers

Order No. DSW-002

Printed in the United States of America

Revision Information for

CONVEX
Tutorial Papers

Edition	Document No.	Description
Eighth	710-011130-001	November 1991. Released with ConvexOS V10.0.
Seventh	710-011130-000	Previously titled *CONVEX UNIX Tutorial Papers*. Released with CONVEX UNIX V9.0, January 1991.
Sixth, Rev 1	710-000850-201	Released with CONVEX UNIX V7.0, November 1988.
Sixth	710-000850-200	Released with CONVEX UNIX V6.0, October 1987.
Fifth	710-000050-000	Released with CONVEX UNIX V5.0, April 1987.
Fourth	710-000250-000	Released with CONVEX UNIX V4.0, August 1986.
Third	710-000250-000	Released with CONVEX UNIX V3.0, October 1985.
Second	710-000250-002	Released with CONVEX UNIX V2.0, July 1985.
First	710-000250-100	Initial release, February 1985.

Contents

Part 4 Supporting tools

Part 5 Miscellaneous material

Bibliography

Using tutorial papers

What are tutorial papers?

ConvexOS Tutorial Papers is a collection of documents that discuss the Berkeley 4.3BSD operating system and the ConvexOS operating system, which is partially derived from 4.3BSD. The selection and organization of these papers are designed to provide an array of introductory and supportive information about ConvexOS and about CONVEX systems and software.

What kind of information can I find?

ConvexOS Tutorial Papers provide

- Basic material for users of CONVEX systems and software.
  An example of this kind of paper is "An Introduction to the C Shell" by W. Joy.
- Supportive documentation to users of CONVEX systems and software.
  An example is "awk—A Pattern Scanning and Processing Language" by Aho, Kernighan, and Weinberger.
- General information about 4.3BSD, most of which is also applicable to CONVEX software. An example is "Mail Refrence Manual" by Kurt Shoens.

How can I find the information I need?

The tutorial papers are organized into the following sections:

Part 1—Introductory papers
Contains information about the C shell, the Bourne shell, Mail, and the `ed` editor.

Part 2—Text editing
Contains material about `ex`, `awk`, and `sed`.

Part 3—Document preparation
Includes information about the -me and -ms macro packages, `style` and `diction`, `nroff`, `troff`, and `tbl`.

Part 4—Supporting tools
Contains material about `lex`, `dc`, `bc`, `make`, `lint`, `yacc`, the -m4 macro processor, and `rcs`.

Part 5—Miscellaneous material
Includes information on security issues, `timed`, and the `man` utility.

The annotated bibliography is a list of third-party documentation that provides further information useful to users of CONVEX systems and software.

How specific to CONVEX systems are the papers?

Because the tutorial papers were not written specifically for ConvexOS or for CONVEX systems, but were written for 4.3BSD, not all the information they contain is specific to ConvexOS. Every effort has been made to ensure that the papers selected are technically accurate or useful for users of CONVEX systems and software.

Ordering documentation

To order the current edition of this or any CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851
USA

Include the order number or the exact title as listed on the front cover.

Part 1
Introductory Papers

An Introduction to the C shell

William Joy
(revised for 4.3BSD by Mark Seiden)

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX† systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shell's capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Additional information includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the UNIX User Reference Manual. The *csh* documentation in section 1 of the manual provides a full description of all features of the shell and is the definitive reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of

† UNIX is a trademark of AT&T Bell Laboratories.

the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution. Jim Kulp added the job control and directory stack primitives and added their documentation to this introduction.

1. Terminal usage of the shell

1.1. The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *programs* are invoked. While it has a set of *builtin* functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system consist of a list of strings or *words* interpreted as a *command name* followed by *arguments*. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to execute it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given as *arguments* to the command itself when it is executed. In this case we specified also the argument *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
     Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a ^D which sent an end-of-file to the mail program. (Here and throughout this document, the notation "^*x*" is to be read "control-*x*" and represents the striking of the *x* key while the control key is held down.) The mail program then echoed the characters 'EOT' and transmitted our message. The characters '% ' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '% ' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file via typing a ^D after which the shell noticed that mail had completed and signaled us that it was ready to read from the terminal again by printing another '% ' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes, it prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal – the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is the delete key (equivalent to '^?') and the kill character is '^U'. Some people prefer to make the erase character the backspace key (equivalent to '^H'). You can make this be true by typing

```
tset -e
```

which tells the program *tset* to set the erase character to tset's default setting for this character (a backspace).

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names, some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '–' (hyphen). Thus the command

```
ls
```

will produce a list of the files in the current *working directory*. The option *–s* is the size option, and

```
ls –s
```

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual section for each command in the UNIX reference manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the date command and the date command prints the date on its standard output. The shell lets us *redirect* the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the *date* command such that its standard output is the file 'now' rather than the terminal. Thus this command places the current date and time into the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to the terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

*Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a CRT, your erase character should be a ^H, as we demonstrated in section 1.1 how this could be set up.

```
date > #now
```

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to use *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading our input. We need not worry about placing shell metacharacters in a letter we are sending via *mail,* or when we are typing in text or data to some other program. Note that the shell is only reading input when it has prompted with '% ' (although we can type our input even before it prompts).

1.5. Input from files; pipelines

We learned above how to *redirect* the *standard output* of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads its input, from the file 'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the sort program would sort lines from its *standard input.* Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a ^D to indicate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of another, i.e. to run the commands in a sequence known as a *pipeline.* For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The *-n* option of sort specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the *ls* command run with the option *-s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *-r* reverse sort option and the *head* command in combination with the previous command doing

```
ls -s | sort -n -r | head -5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have

run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The notation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX *pathnames* consist of a number of *components* separated by '/'. Each component except the last names a directory in which the next component resides, in effect specifying the *path* of directories to follow to reach the file. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. A *pathname* that begins with a slash is said to be an *absolute* pathname since it is specified from the absolute top of the entire directory hierarchy of the system (the *root*). *Pathnames* which do not begin with '/' are interpreted as starting in the current *working directory*, which is, by default, your *home* directory and can be changed dynamically by the *cd* change directory command. Such pathnames are said to be *relative* to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each *component* of the pathname. If the pathname contains no slashes at all then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames have no relation to the working directory.

Most filenames consist of a number of alphanumeric characters and '.'s (periods). In fact, all printing characters except '/' (slash) may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' (period) is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a *base* portion of a name (a base portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the notation

```
prog.*
```

This expression is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are alphabetically sorted and placed in the *argument list* of the command. Thus the command

```
echo prog.*
```

will echo the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in sorted order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by *filename expansion* of the one input word.

Other notations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

```
prog.[co]
```

will match

```
prog.c prog.o
```

in the example above. We can also place two characters around a '–' in this notation to denote a range. Thus

```
chap.[1–5]
```

might match files

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list)* contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

```
No match.
```

and does not execute the command.

Another very important point is that files with the character '.' at the beginning are treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the working directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' (tilde) followed by another user's login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a convenient way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department VAX machine, where this document was prepared. This can be very useful if you have used *cd* to change to another directory and have found a file you wish to copy using *cp*. If I give the command

```
cp thatfile ~
```

the shell will expand this command to

```
cp thatfile /usr/bill
```

since my home directory is /usr/bill.

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of words which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.2, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character '*'. It will either echo an sorted list of filenames in the current *working directory,* or print the message 'No match' if there are no files in the working directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.' or '–' in an argument word to a command is to enclose it with single quotation characters '', i.e.

```
echo '*'
```

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* by placing it within '' characters. It and the character '' itself can be preceded by a single '\' to prevent their special meaning. Thus

```
echo \'\!
```

prints

```
'!
```

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

```
echo \''*'
```

which prints

```
'*
```

since the first '\' escaped the first '' and the '*' was enclosed between '' characters.

1.8. Terminating commands

When you are executing a command and the shell is waiting for it to complete there are several ways to force it to stop. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT *signal* to the *cat* command by typing ^C on your terminal.* Since *cat* does not take any precautions to avoid or otherwise handle this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has terminated and prompts you again with '% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it handles INTERRUPT signals and chooses to continue to execute commands rather than terminating like *cat* did, which would have the effect of logging you out.

*On some older Unix systems the DEL or RUBOUT key has the same effect. "stty all" will tell you the INTR key value.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we typed a ^D which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many ^D's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a ^D. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor program. We could also have done

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

Another possibility for stopping a command is to suspend its execution temporarily, with the possibility of continuing execution later. This is done by sending a STOP signal via typing a ^Z. This signal causes all commands running on the terminal (usually one but more if a pipeline is executing) to become suspended. The shell notices that the command(s) have been suspended, types 'Stopped' and then prompts for a new command. The previously executing command has been suspended, but otherwise unaffected by the STOP signal. Any other commands can be executed while the original command remains suspended. The suspended command can be continued using the *fg* command with no arguments. The shell will then retype the command to remind you which command is being continued, and cause the command to resume execution. Unless any input files in use by the suspended command have been changed in the meantime, the suspension has no effect whatsoever on the execution of the command. This feature can be very useful during editing, when you need to look at another file before continuing. An example of command suspension follows.

```
% mail harold
Someone just copied a big file into my directory and its name is
^Z
Stopped
% ls
funnyfile
prog.c
prog.o
% jobs
[1] + Stopped                 mail harold
% fg
mail harold
funnyfile. Do you know who did it?
EOT
%
```

In this example someone was sending a message to Harold and forgot the name of the file he wanted to mention. The mail command was suspended by typing ^Z. When the shell noticed that the mail program was suspended, it typed 'Stopped' and prompted for a new command. Then the *ls* command was typed to find out the name of the file. The *jobs* command was run to find out which command was suspended. At this time the *fg* command was typed to continue

execution of the mail program. Input to the mail program was then continued and ended with a ^D which indicated the end of the message at which time the mail program typed EOT. The *jobs* command will show which commands are suspended. The ^Z should only be typed at the beginning of a line since everything typed on the current line is discarded when a signal is sent from the keyboard. This also happens on INTERRUPT, and QUIT signals. More information on suspending jobs and controlling them is given in section 2.6.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, sent by typing a ^\. This will usually provoke the shell to produce a message like:

```
Quit (Core dumped)
```

indicating that a file 'core' has been created containing information about the running program's state when it terminated due to the QUIT signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* command. See section 2.6 for an example.

If you want to examine the output of a command without having it move off the screen as the output of the

```
cat /etc/passwd
```

command will, you can use the command

```
more /etc/passwd
```

The *more* program pauses after each complete screenful and types '—More—' at which point you can hit a space to get another screenful, a return to get another line, a '?' to get some help on other commands, or a 'q' to end the *more* program. You can also use more as a filter, i.e.

```
cat /etc/passwd | more
```

works just like the more simple more command above.

For stopping output of commands not involving *more* you can use the ^S key to stop the typeout. The typeout will resume when you hit ^Q or any other key, but ^Q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type ^S and ^Q fast enough to paginate the output nicely, and a program like *more* is usually used.

An additional possibility is to use the ^O flush output character; when this character is typed, all output from the current command is thrown away (quickly) until the next input read occurs or until the next shell prompt. This can be used to allow a command to complete without having to suffer through the output on a slow terminal; ^O is a toggle, so flushing can be turned off by typing ^O again while output is being flushed.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

```
chsh myname /bin/csh
```

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it takes effect at next login.** You are now ready to try using *csh*.

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is started by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login shell*, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
        'set noglob ; eval `tset -s -m dialup:c100rv4pna -m plugboard:?hp2621nl *` ';
ts; stty intr ^C kill ^U crt
set time=15 history=10
msgs -f
if (-e $mail) then
        echo "${prompt}mail"
        mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit ^D. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there. An alternative to this is to put the command

```
biff y
```

in place of this *set;* this will cause me to be notified immediately when mail arrives, and to be shown the first few lines of the new message.

Next I set the shell variable 'time' to '15' causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable 'history' is set to 10 indicating that I want the shell to remember the last 10 commands I type in its *history list*, (described later).

I create an *alias* "ts" which executes a *tset*(1) command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal which I usually use when not on a hardwired port. I then execute "ts" and also use the *stty* command to change the interrupt character to ^C and the line kill character to ^U.

I then run the 'msgs' program, which provides me with any system messages which I have not seen before; the '-f' option here prevents it from telling me anything if there are no new messages. Finally, if my mailbox file exists, then I run the 'mail' program to process my mail.

When the 'mail' and 'msgs' programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with '% '. When I log off (by giving the *logout* command) the shell will print 'logout' and execute commands from the file '.logout' if it exists in my home directory. After that the shell will terminate and UNIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the 'logout' message the shell is committed to terminating and will take no further

input from my terminal.

2.2. Shell variables

The shell maintains a set of *variables*. We saw above the variables *history* and *time* which had values '10' and '15'. In fact, each shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the shell searches for commands. The *set* command with no arguments shows the value of all variables currently defined (we usually say *set)* in the shell. The default value for path will be shown by *set* to be

```
% set
argv    ()
cwd     /usr/bill
home    /usr/bill
path    (. /usr/ucb /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
term    c100rv4pna
user    bill
%
```

This output indicates that the variable path points to the current directory '.' and then '/usr/ucb', '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Commands developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish that all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

† Another directory that might interest you is /usr/new, which contains many useful user-contributed programs provided with Berkeley Unix.

rehash

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory '.' on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable *home* which shows your home directory, *cwd* which contains your current working directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset*. Thus to set this variable you simply do

set ignoreeof

and to unset it do

unset ignoreeof

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail*. The metasyntax

> filename

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

set noclobber

in your *.login* file. Then trying to do

date > now

would cause a diagnostic if 'now' existed already. You could type

date >! now

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.†

2.3. The shell's history list

The shell can maintain a *history list* into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell. In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '!$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation metacharacter, and the '$' stands for the last argument, by analogy to '$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other

†The space between the '!' and the word 'now' is critical here, as '!now' would be an invocation of the *history* mechanism, and have a totally different effect.

```
% cat bug.c
main()

{
        printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
        printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
        printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill       3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill       3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ^spp^ssp
num bug.c | ssp
    1  main()
    3  {
    4          printf("hello\n");
    5  }
% !! | lpr
num bug.c | ssp | lpr
%
```

commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '–o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls –l' command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a numbered listing of the program we ran the 'num' command on the file 'bug.c'. In order to compress out blank lines in the output of 'num' we ran the output through the filter 'ssp', but misspelled it as spp. To correct this we used a shell substitute, placing the old text and new text between '^' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. The *history* command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmer's Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as *cd* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

```
mail bill
```

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '–s'. We can do

```
alias ls ls –s
```

or even

```
alias dir ls –s
```

creating a new command syntax 'dir' which does an 'ls –s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls –s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in '' characters to prevent most substitutions from occurring and the character ';' from being recognized as a metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!^ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

Warning: The shell currently reads the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

The '>&' here tells the shell to route both the diagnostic output and the standard output into 'file'. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.‡

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.†

---

‡ A command of the form

```
command >&! file
```

exists, and is used when *noclobber* is set and *file* already exists.

† If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for file to not exist when *noclobber* is set.

2.6. Jobs; Background, Foreground, or Suspended

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single *job* is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the metacharacter '&' is typed at the end of the commands, then the job is started as a *background* job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs *in the background* at the same time that normal jobs, called *foreground* jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the *du* program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file 'usage' and return immediately with a prompt for the next command without out waiting for *du* to finish. The *du* program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. When a background job terminates, a message is typed by the shell just before the next prompt telling you that the job has completed. In the following example the *du* job finishes sometime during the execution of the *mail* command and its completion is reported just before the prompt after the *mail* job is finished.

```
% du > usage &
[1] 503
% mail bill
How do you know when a background job is finished?
EOT
[1] - Done                    du > usage
%
```

If the job did not terminate normally the 'Done' message might say something else like 'Killed'. If you want the terminations of background jobs to be reported at the time they occur (possibly interrupting the output of other foreground jobs), you can set the *notify* variable. In the previous example this would mean that the 'Done' message might have come right in the middle of the message to Bill. Background jobs are unaffected by any signals from the keyboard like the STOP, INTERRUPT, or QUIT signals mentioned earlier.

Jobs are recorded in a table inside the shell until they terminate. In this table, the shell remembers the command names, arguments and the *process numbers* of all commands in the job as well as the working directory where the job was started. Each job in the table is either running *in the foreground* with the shell waiting for it to terminate, running *in the background,* or *suspended.* Only one job can be running in the foreground at one time, but several jobs can be suspended or running in the background at once. As each job is started, it is assigned a small identifying number called the *job number* which can be used later to refer to the job in the commands described below. Job numbers remain the same until the job terminates and then are reused.

When a job is started in the backgound using '&', its number, as well as the process numbers of all its (top level) commands, is typed by the shell before prompting you for another command. For example,

```
% ls -s | sort -n > usage &
[2] 2034 2035
%
```

runs the 'ls' program with the '-s' options, pipes this output into the 'sort' program with the '-n' option which puts its output into the file 'usage'. Since the '&' was at the end of the line, these two programs were started together as a background job. After starting the job, the shell prints the job number in brackets (2 in this case) followed by the process number of each program started in the job. Then the shell immediates prompts for a new command, leaving the job running simultaneously.

As mentioned in section 1.8, foreground jobs become *suspended* by typing ^Z which sends a STOP signal to the currently running foreground job. A background job can become suspended by using the *stop* command described below. When jobs are suspended they merely stop any further progress until started again, either in the foreground or the backgound. The shell notices when a job becomes stopped and reports this fact, much like it reports the termination of background jobs. For foreground jobs this looks like

```
% du > usage
^Z
Stopped
%
```

'Stopped' message is typed by the shell when it notices that the *du* program stopped. For background jobs, using the *stop* command, it is

```
% sort usage &
[1] 2345
% stop %1
[1] + Stopped (signal)          sort usage
%
```

Suspending foreground jobs can be very useful when you need to temporarily change what you are doing (execute other commands) and then return to the suspended job. Also, foreground jobs can be suspended and then continued as background jobs using the *bg* command, allowing you to continue other work and stop waiting for the foreground job to finish. Thus

```
% du > usage
^Z
Stopped
% bg
[1] du > usage &
%
```

starts 'du' in the foreground, stops it before it finishes, then continues it in the background allowing more foreground commands to be executed. This is especially helpful when a foreground job ends up taking longer than you expected and you wish you had started it in the backgound in the beginning.

All *job control* commands can take an argument that identifies a particular job. All job name arguments begin with the character '%', since some of the job control commands also accept process numbers (printed by the *ps* command.) The default job (when no argument is given) is called the *current* job and is identified by a '+' in the output of the *jobs* command, which shows you which jobs you have. When only one job is stopped or running in the background (the usual case) it is always the current job thus no argument is needed. If a job is stopped while running in the foreground it becomes the *current* job and the existing current job becomes the *previous* job - identified by a '-' in the output of *jobs*. When the current job terminates, the previous job becomes the current job. When given, the argument is either '%-' (indicating the previous job); '%#', where # is the job number; '%pref' where pref is some unique

prefix of the command name and arguments of one of the jobs; or '%?' followed by some string found in only one of the jobs.

The *jobs* command types the table of jobs, giving the job number, commands and status ('Stopped' or 'Running') of each backgound or suspended job. With the '–l' option the process numbers are also typed.

```
% du > usage &
[1] 3398
% ls –s | sort –n > myfile &
[2] 3405
% mail bill
^Z
Stopped
% jobs
[1] – Running                du > usage
[2]   Running                ls –s | sort –n > myfile
[3] + Stopped                mail bill
% fg %ls
ls –s | sort –n > myfile
% more myfile
```

The *fg* command runs a suspended or background job in the foreground. It is used to restart a previously suspended job or change a background job to run in the foreground (allowing signals or input from the terminal). In the above example we used *fg* to change the 'ls' job from the background to the foreground since we wanted to wait for it to finish before looking at its output file. The *bg* command runs a suspended job in the background. It is usually used after stopping the currently running foreground job with the STOP signal. The combination of the STOP signal and the *bg* command changes a foreground job into a background job. The *stop* command suspends a background job.

The *kill* command terminates a background or suspended job immediately. In addition to jobs, it may be given process numbers as arguments, as printed by *ps*. Thus, in the example above, the running *du* command could have been terminated by the command

```
% kill %1
[1]  Terminated              du > usage
%
```

The *notify* command (not the variable mentioned earlier) indicates that the termination of a specific job should be reported at the time it finishes instead of waiting for the next prompt.

If a job running in the background tries to read input from the terminal it is automatically stopped. When such a job is then run in the foreground, input can be given to the job. If desired, the job can be run in the background again until it requests input again. This is illustrated in the following sequence where the 's' command in the text editor might take a long time.

```
% ed bigfile
120000
1,$s/thisword/thatword/
^Z
Stopped
% bg
[1] ed bigfile &
%
 . . . some foreground commands
[1] Stopped (tty input)      ed bigfile
% fg
```

```
ed bigfile
w
120000
q
%
```

So after the 's' command was issued, the 'ed' job was stopped with ^Z and then put in the background using *bg*. Some time later when the 's' command was finished, *ed* tried to read another command and was stopped because jobs in the backgound cannot read from the terminal. The *fg* command returned the 'ed' job to the foreground where it could once again accept commands from the terminal.

The command

```
stty tostop
```

causes all background jobs run on your terminal to stop when they are about to write output to the terminal. This prevents messages from background jobs from interrupting foreground job output and allows you to run a job in the background without losing terminal output. It also can be used for interactive programs that sometimes have long periods without interaction. Thus each time it outputs a prompt for more input it will stop before the prompt. It can then be run in the foreground using *fg*, more input can be given and, if necessary stopped and returned to the background. This *stty* command might be a good thing to put in your *.login* file if you do not like output from background jobs interrupting your work. It also can reduce the need for redirecting the output of background jobs if the output is not very big:

```
% stty tostop
% wc hugefile &
[1] 10387
% ed text
. . . some time later
q
[1] Stopped (tty output)     wc hugefile
% fg wc
wc hugefile
  13371   30123   302577
% stty -tostop
```

Thus after some time the 'wc' command, which counts the lines, words and characters in a file, had one line of output. When it tried to write this to the terminal it stopped. By restarting it in the foreground we allowed it to write on the terminal exactly when we were ready to look at its output. Programs which attempt to change the mode of the terminal will also block, whether or not *tostop* is set, when they are not in the foreground, as it would be very unpleasant to have a background job change the state of the terminal.

Since the *jobs* command only prints jobs started in the currently executing shell, it knows nothing about background jobs started in other login sessions or within shell files. The *ps* can be used in this case to find out about background jobs not started in the current shell.

2.7. Working Directories

As mentioned in section 1.6, the shell is always in a particular *working directory*. The 'change directory' command *chdir* (its short form *cd* may also be used) changes the working directory of the shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The 'make directory' command, *mkdir*, creates a new directory. The *pwd* ('print working directory') command reports the absolute pathname of the working directory of the shell, that is, the directory you are located in. Thus in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory *newpaper.* where, for example, he might place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your 'home' login directory by doing just

```
cd
```

with no arguments. The name '..' always means the directory above the current one in the hierarchy, thus

```
cd ..
```

changes the shell's working directory to the one directly above the current one. The name '..' can be used in any pathname, thus,

```
cd ../programs
```

means change to the directory 'programs' contained in the directory above the current one. If you have several directories for different projects under, say, your home directory, this shorthand notation permits you to switch easily between them.

The shell always remembers the pathname of its current working directory in the variable *cwd.* The shell can also be requested to remember the previous directory when you change to a new working directory. If the 'push directory' command *pushd* is used in place of the *cd* command, the shell saves the name of the current working directory on a *directory stack* before changing to the new one. You can see this list at any time by typing the 'directories' command *dirs.*

```
% pushd newpaper/references
~/newpaper/references ~
% pushd /usr/lib/tmac
/usr/lib/tmac ~/newpaper/references ~
% dirs
/usr/lib/tmac ~/newpaper/references ~
% popd
~/newpaper/references ~
% popd
~
%
```

The list is printed in a horizontal line, reading left to right, with a tilde (~) as shorthand for your home directory—in this case '/usr/bill'. The directory stack is printed whenever there is more than one entry on it and it changes. It is also printed by a *dirs* command. *Dirs* is usually faster and more informative than *pwd* since it shows the current working directory as well as any other directories remembered in the stack.

The *pushd* command with no argument alternates the current directory with the first directory in the list. The 'pop directory' *popd* command without an argument returns you to the directory you were in prior to the current one, discarding the previous current directory from the stack (forgetting it). Typing *popd* several times in a series takes you backward through the directories you had been in (changed to) by *pushd* command. There are other options to *pushd* and *popd* to manipulate the contents of the directory stack and to change to directories not at the top

of the stack; see the *csh* manual page for details.

Since the shell remembers the working directory in which each job was started, it warns you when you might be confused by restarting a job in the foreground which has a different working directory than the current working directory of the shell. Thus if you start a background job, then change the shell's working directory and then cause the background job to run in the foreground, the shell warns you that the working directory of the currently running foreground job is different from that of the shell.

```
% dirs –l
/mnt/bill
% cd myproject
% dirs
~/myproject
% ed prog.c
1143
^Z
Stopped
% cd ..
% ls
myproject
textfile
% fg
ed prog.c (wd: ~/myproject)
```

This way the shell warns you when there is an implied change of working directory, even though no cd command was issued. In the above example the 'ed' job was still in '/mnt/bill/project' even though the shell had changed to '/mnt/bill'. A similar warning is given when such a foreground job terminates or is suspended (using the STOP signal) since the return to the shell again implies a change of working directory.

```
% fg
ed prog.c (wd: ~/myproject)
 . . . after some editing
q
(wd now: ~)
%
```

These messages are sometimes confusing if you use programs that change their own working directories, since the shell only remembers which directory a job is started in, and assumes it stays there. The '–l' option of *jobs* will type the working directory of suspended or background jobs when it is different from the current working directory of the shell.

2.8. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., 'ls'.

The *echo* command prints its arguments. It is often used in *shell scripts* or as an interactive command to see what filename expansions will produce.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using

the contextual mechanisms introduced above. There is also a shell variable called *prompt*. By placing a '!' character in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt='\! % '
```

Note that the '!' character had to be *escaped* here even within '' characters.

The *limit* command is used to restrict use of resources. With no arguments it prints the current limitations:

```
cputime      unlimited
filesize     unlimited
datasize     5616 kbytes
stacksize    512 kbytes
coredumpsize unlimited
```

Limits can be set, e.g.:

```
limit coredumpsize 128k
```

Most reasonable units abbreviations will work; see the *csh* manual page for more details.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

```
repeat 5 cat one >> five
```

The *setenv* command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable TERM to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect right away.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8% 2+1k 3+2io 1pf+0w
% time wc /etc/rc /usr/bill/rc
    52   178  1347 /etc/rc
    52   178  1347 /usr/bill/rc
   104   356  2694 total
0.1u 0.1s 0:00 13% 3+3k 5+3io 7pf+0w
%
```

indicates that the *cp* command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01), there was an average memory usage of 2k bytes of program space and 1k bytes of data space over the cpu time involved (2+1k); the program did three disk reads and two disk writes (3+2io), and took one page fault and was not swapped (1pf+0w). The word count command *wc* on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage '13%' indicates that over the period when it was active the command 'wc' used an average of 13 percent of the available CPU cycles of the machine.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell, and *unsetenv* removes variables from the environment.

2.9. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you you should look through the rest of this document and the csh manual pages (section1) to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile*. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *csh* command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character '$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. It is the fundamental mechanism used

for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of *argv* or in the example above 'a'. Similarly

```
$argv[$#argv]
```

would give 'c', and

```
$argv[1-2]
```

would give 'a b'. Other notations useful in shell scripts are

```
$n
```

where *n* is an integer as a shorthand for

```
$argv[n]
```

the *n th* parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt 'yes or no?' without a newline and then read the answer into the

variable 'a'. In this case '$#a' would be '0' if either a blank line or end-of-file (^D) was typed.

One minor difference between '$*n*' and '$argv[*n*]' should be noted here. The form '$argv[*n*]' will yield an error if *n* is not in the range '1-$#argv' while '$n' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form 'n-'; if there are less than *n* components of the given variable then no words are substituted. A range of the form 'm-n' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations. The special operators '=~' and '!~' are similar to '==' and '!=' except that the string on the right side can have pattern matching characters (like *, ? or []) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

-? filename

where '?' is replace by a number of single characters. For instance the expression primitive

-e filename

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '$status' examined in the next command. Since '$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

†The following two formats are not currently acceptable to the shell:

```
if ( expression )               # Won't work!
then
        command
        ...
endif
```

and

```
if ( expression ) then command endif            # Won't work
```

The shell does have another form of the if statement of the form

if (expression) **command**

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else-if* pairs followed by a single *else* and an *endif*, e.g.:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in shell scripts is the ':' modifier. We can use the modifier ':r' here to extract a root of a filename or ':e' to extract the *extension*. Thus if the variable *i* has the value '/mnt/foo.bar' then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the ':r' modifier strips off the trailing '.bar' and the the ':e' modifier leaves only the 'bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual pages in the User's Reference Manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism.‡ Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using '' or '\' to place it in an argument word.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

‡ It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

```
while ( expression )
    commands
end
```

and

```
switch ( word )

case str1:
    commands
    breaksw

...

case strn:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

```
loop:
    commands
    goto loop
```

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under UNIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

```
% cat deblank
# deblank — remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly ''EOF''. The fact that the 'EOF' is enclosed in '' characters, i.e. quoted, causes the shell to not perform variable

substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,$' in our editor script we needed to insure that this '$' was not variable substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/^[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do an *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *-v* and *-x* command line options can be used to help trace the actions of the shell. The *-n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *csh* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '"' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as '`' does.

4. Other, less commonly used, shell features

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell one could have issued the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with '? ' when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within '`' characters is converted by the shell to a list of words. You can also place the '`' quoted string within '"' characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier ':x' exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

mkdir ~/{hdrs,retrofit,csh}

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}

4.3. Command substitution

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

set pwd=`pwd`

to save the current directory in the variable *pwd* or to do

ex `grep -l TRACE *.c`

to run the editor *ex* supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the csh(1) manual section for a list of these options.

*Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the shell manual section for full details.

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX User Reference manual in section 1. You can look at an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

.
: Your current directory has the name '.' as well as the name printed by the command *pwd;* see also *dirs*. The current directory '.' is usually the first *component* of the search path contained in the variable *path*, thus commands which are in '.' are found first (2.2). The character '.' is also used in separating *components* of filenames (1.6). The character '.' at the beginning of a *component* of a *pathname* is treated specially and not matched by the *filename expansion* metacharacters '?', '*', and '[' ']' pairs (1.6).

..
: Each directory has a file '..' in it which is a reference to its parent directory. After changing into the directory with *chdir*, i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by *pwd* (2.7).

a.out
: Compilers which create executable images create them, by default, in the file *a.out.* for historical reasons (2.3).

absolute pathname
: A *pathname* which begins with a '/' is *absolute* since it specifies the *path* of directories from the beginning of the entire directory system – called the *root* directory. *Pathnames* which are not *absolute* are called *relative* (see definition of *relative pathname*) (1.6).

alias
: An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes *aliases* and can print their current values. The command *unalias* is used to remove *aliases* (2.4).

argument
: Commands in UNIX receive a list of *argument* words. Thus the command

```
echo a b c
```

consists of the *command name* 'echo' and three *argument* words 'a', 'b' and 'c'. The set of *arguments* after the *command name* is said to be the *argument list* of the command (1.1).

argv
: The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).

background
: Commands started without waiting for them to complete are called *background* commands (2.6).

base
: A filename is sometimes thought of as consisting of a *base* part, before any '.' character, and an *extension* – the part after the '.'. See *filename* and *extension* (1.6) and basename (1).

bg — The *bg* command causes a *suspended* job to continue execution in the *background* (2.6).

bin — A directory containing binaries of programs and shell scripts to be executed is typically called a *bin* directory. The standard system *bin* directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a *component* of the variable *path*.

break — *Break* is a builtin command used to exit from loops within the control structure of the shell (3.7).

breaksw — The *breaksw* builtin command is used to exit from a *switch* control structure, like a *break* exits from loops (3.7).

builtin — A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in *bin* directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

case — A *case* command is used as a label in a *switch* statement in the shell's control structure, similar to that of the language C. Details are given in the shell documentation 'csh (1)' (3.7).

cat — The *cat* program catenates a list of specified files on the *standard output*. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).

cd — The *cd* command is used to change the *working directory*. With no arguments, *cd* changes your *working directory* to be your *home* directory (2.4, 2.7).

chdir — The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.

chsh — The *chsh* command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing

```
chsh your-login-name /bin/csh
```

Thus I would do

```
chsh bill /bin/csh
```

It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).

cmp — *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.

command — A function performed by the system, either by the shell (a builtin *command*) or by a program residing in a file in a directory within the UNIX system, is called a *command* (1.1).

command name — When a command is issued, it consists of a *command name*, which is the first word of the command, followed by arguments. The convention on UNIX is that the first word of a command names the function to be performed (1.1).

command substitution — The replacement of a command enclosed in '`' characters by the text output by

that command is called *command substitution* (4.3).

component — A part of a *pathname* between '/' characters is called a *component* of that *pathname*. A variable which has multiple strings as value is said to have several *components*; each string is a *component* of the variable.

continue — A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).

control- — Certain special characters, called *control* characters, are produced by holding down the CONTROL key on your terminal and simultaneously pressing another character, much like the SHIFT key is used to produce upper case characters. Thus *control-c* is produced by holding down the CONTROL key while pressing the 'c' key. Usually UNIX prints an caret (^) followed by the corresponding letter when you type a *control* character (e.g. '^C' for *control-c* (1.8).

core dump — When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This *core dump* can be examined with the system debugger 'adb (1)' or 'sdb (1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form

Illegal instruction (core dumped)

(where 'Illegal instruction' is only one of several possible messages), you should report this to the author of the program or a system administrator, saving the 'core' file.

cp — The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (1.6).

csh — The name of the shell program that this document describes.

.cshrc — The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).

cwd — The *cwd* variable in the shell holds the *absolute pathname* of the current *working directory*. It is changed by the shell whenever your current *working directory* changes and should not be changed otherwise (2.2).

date — The *date* command prints the current date and time (1.3).

debugging — *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell *debugging* (4.4).

default: — The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).

DELETE — The DELETE or RUBOUT key on the terminal normally causes an interrupt to be sent to the current job. Many users change the interrupt character to be ^C.

detached — A command that continues running in the *background* after you logout is said to be *detached*.

diagnostic — An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the *standard output*, since that is often directed away from the terminal (1.3, 1.5). Error messsages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus *diagnostics* will usually appear on the terminal (2.5).

directory — A structure which contains files. At any time you are in one particular *directory* whose names can be printed by the command *pwd*. The *chdir* command will change you to another *directory*, and make the files in that *directory*

visible. The *directory* in which you are when you first login is your *home* directory (1.1, 2.7).

directory stack — The shell saves the names of previous *working directories* in the *directory stack* when you change your current *working directory* via the *pushd* command. The *directory stack* can be printed by using the *dirs* command, which includes your current *working directory* as the first directory name on the left (2.7).

dirs — The *dirs* command prints the shell's *directory stack* (2.7).

du — The *du* command is a program (described in 'du (1)') which prints the number of disk blocks is all directories below and including your current *working directory* (2.6).

echo — The *echo* command prints its arguments (1.6, 3.6).

else — The *else* command is part of the 'if-then-else-endif' control command construct (3.6).

endif — If an *if* statement is ended with the word *then*, all lines following the *if* up to a line starting with the word *endif* or *else* are executed if the condition between parentheses after the *if* is true (3.6).

EOF — An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an *end-of-file* when the command sending them input completes. Most commands terminate when they receive an *end-of-file*. The shell has an option to ignore *end-of-file* from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).

escape — A character '\' used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ *escapes* '*' (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be *suspended*. Most systems use control-s to stop the output and control-q to start it.

/etc/passwd — This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters (1.8). You can look at this file by saying

```
cat /etc/passwd
```

The commands *finger* and *grep* are often used to search for information in this file. See 'finger (1)', 'passwd(5)', and 'grep (1)' for more details.

exit — The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).

exit status — A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script to give a non-zero *exit status* (3.6).

expansion — The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion*. Thus the replacement of

the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. *Expansions* are also referred to as *substitutions* (1.6, 3.4, 4.2).

expressions — *Expressions* are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell *expressions* are those of the language C (3.5).

extension — Filenames often consist of a *base* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same *root* name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '–me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).

fg — The *job control* command *fg* is used to run a *background* or *suspended* job in the *foreground* (1.8, 2.6).

filename — Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most *filenames* do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the *base* portion of the *filename* from an *extension* (1.6).

filename expansion — *Filename expansion* uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using *filename expansion* it is easy to name all the files in the current directory, or all files which have a common *root* name. Other *filename expansion* mechanisms use the metacharacter '~' and allow files in other users' directories to be named easily (1.6, 4.2).

flag — Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters preceded by the character '–' (1.2`. Thus the *ls* (list files) command has an option '–s' to list the sizes of files. This is specified

```
ls –s
```

foreach — The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).

foreground — When commands are executing in the normal way such that the shell is waiting for them to finish before prompting for another command they are said to be *foreground jobs* or *running in the foreground*. This is as opposed to *background*. *Foreground* jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard (1.8, 2.6).

goto — The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).

grep — The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file */etc/passwd* which contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed (1)' and 'ex (1)'. *Grep* stands for 'globally find *regular expression* and print' (2.4).

head — The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes

useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5).
Head is also used to describe the part of a *pathname* before and including the last '/' character. The *tail* of a *pathname* is the part after the last '/'. The ':h' and ':t' modifiers allow the *head* or *tail* of a *pathname* stored in a shell variable to be used (3.6).

history — The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (2.3).

home directory — Each user has a *home directory*, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first login. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the *home directories* of other users in forming filenames using a *filename expansion* notation and the character '~' (1.6).

if — A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).

ignoreeof — Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2).

input — Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for *input* from their *standard input* which is, by default, the terminal. This *standard input* can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in *pipelines* will read from the output of the previous command in the *pipeline*. The leftmost command in a *pipeline* reads from the terminal if you neither redirect its *input* nor give it a filename to use as *standard input*. Special mechanisms exist for supplying input to commands in shell scripts (1.5, 3.8).

interrupt — An *interrupt* is a signal to a program that is generated by typing ^C. (On older versions of UNIX the RUBOUT or DELETE key were used for this purpose.) It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an *interrupt* in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to *interrupts*. The shell often wakes up when you hit *interrupt* because many commands die when they receive an *interrupt* (1.8, 3.9).

job — One or more commands typed on the same input line separated by '|' or ';' characters are run together and are called a *job*. Simple commands run by themselves without any '|' or ';' characters are the simplest *jobs*. *Jobs* are classified as *foreground*, *background*, or *suspended* (2.6).

job control — The builtin functions that control the execution of jobs are called *job control* commands. These are *bg, fg, stop, kill* (2.6).

job number — When each job is started it is assigned a small number called a *job number* which is printed next to the job in the output of the *jobs* command. This number, preceded by a '%' character, can be used as an argument to *job control* commands to indicate a specific job (2.6).

jobs — The *jobs* command prints a table showing jobs that are either running in the *background* or are *suspended* (2.6).

kill — A command which sends a signal to a job causing it to terminate (2.6).

.login — The file *.login* in your *home* directory is read by the shell each time you login to UNIX and the commands there are executed. There are a number of commands which are usefully placed here, especially *set* commands to the shell itself (2.1).

login shell — The shell that is started on your terminal when you login is called your *login shell*. It is different from other shells which you may run (e.g. on shell scripts) in that it reads the *.login* file before reading commands from the terminal and it reads the *.logout* file after you logout (2.1).

logout — The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an *end-of-file,* but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.8).

.logout — When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.

lpr — The command *lpr* is the line printer daemon. The standard input of *lpr* spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).

ls — The *ls* (list files) command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).

mail — The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.1), whether they are logged on or not.

make — The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).

makefile — The file containing commands for *make* is called *makefile* or *Makefile* (3.2).

manual — The *manual* often referred to is the 'UNIX manual'. It contains 8 numbered sections with a description of each UNIX program (section 1), system call (section 2), subroutine (section 3), device (section 4), special data structure (section 5), game (section 6), miscellaneous item (section 7) and system administration program (section 8). There are also supplementary documents (tutorials and reference guides) for individual programs which require explanation in more detail. An online version of the *manual* is accessible through the *man* command. Its documentation can be obtained online via

```
man man
```

If you can't decide what manual page to look in, try the *apropos*(1) command. The supplementary documents are in subdirectories of /usr/doc.

metacharacter — Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a *metacharacter* is the character '>' which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted *metacharacters* form separate words (1.4). The appendix to this user's manual lists the

metacharacters in groups by their function.

mkdir — The *mkdir* command is used to create a new directory.

modifier — Substitutions with the *history* mechanism, keyed by the character '!' or of variables using the metacharacter '$', are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the *modifier* itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).

more — The program *more* writes a file on your terminal allowing you to control how much text is displayed at a time. *More* can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way of viewing a file (1.8).

noclobber — The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5).

noglob — The shell variable *noglob* is set to suppress the *filename expansion* of arguments containing the metacharacters '~', '*', '?', '[' and ']' (3.6).

notify — The *notify* command tells the shell to report on the termination of a specific *background job* at the exact time it occurs as opposed to waiting until just before the next prompt to report the termination. The *notify* variable, if set, causes the shell to always report the termination of *background* jobs exactly when they occur (2.6).

onintr — The *onintr* command is built into the shell and is used to control the action of a shell command script when an *interrupt* signal is received (3.9).

output — Many commands in UNIX result in some lines of text which are called their *output*. This *output* is usually placed on what is known as the *standard output* which is normally connected to the user's terminal. The shell has a syntax using the metacharacter '>' for redirecting the *standard output* of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|' it is also possible for the *standard output* of one command to become the *standard input* of another command (1.5). Certain commands such as the line printer daemon *p* do not place their results on the *standard output* but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another user's terminal rather than its *standard output* (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the *standard output* has been sent to a file or another command, but it is possible to direct error diagnostics along with *standard output* using a special metanotation (2.5).

path — The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

```
path (. /usr/ucb /bin /usr/bin)
```

the shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' permission set. This is normally true because a command of the form

chmod 755 script

was executed to turn this execute permission on (3.3). If you add new commands to a directory in the *path*, you should issue the command *rehash* (2.2).

pathname — A list of names, separated by '/' characters, forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next *component* file resides. *Pathnames* which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other *pathnames* are interpreted relative to the current directory as reported by *pwd*. The last component of a *pathname* may name a directory, but usually names a file.

pipeline — A group of commands which are connected together, the *standard output* of each connected to the *standard input* of the next, is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|' (1.5, 2.3).

popd — The *popd* command changes the shell's *working directory* to the directory you most recently left using the *pushd* command. It returns to the directory without having to type its name, forgetting the name of the current *working directory* before doing so (2.7).

port — The part of a computer system to which each terminal is connected is called a *port*. Usually the system has a fixed number of *ports*, some of which are connected to telephone lines for dial-up access, and some of which are permanently wired directly to specific terminals.

pr — The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

printenv — The *printenv* command is used to print the current setting of variables in the environment (2.8).

process — An instance of a running program is called a *process* (2.6). UNIX assigns each *process* a unique number when it is started – called the *process number*. *Process numbers* can be used to stop individual *processes* using the *kill* or *stop* commands when the *processes* are part of a detached *background* job.

program — Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a *program*.

prompt — Many programs will print a *prompt* on the terminal when they expect input. Thus the editor 'ex (1)' will print a ':' when it expects input. The shell *prompts* for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shell's main *prompt*. This is mostly used when debugging the shell (2.8).

pushd — The *pushd* command, which means 'push directory', changes the shell's *working directory* and also remembers the current *working directory* before the change is made, allowing you to return to the same directory via the *popd* command later without retyping its name (2.7).

ps — The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, an indication of the state of the process (whether it is running, stopped, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the *csh* you use to run the *ps* command, are not normally shown in the output.

pwd — The *pwd* command prints the full *pathname* of the current *working directory*. The *dirs* builtin command is usually a better and faster choice.

quit — The *quit* signal, generated by a control-\, is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).

quotation — The process by which metacharacters are prevented their special meaning, usually by using the character ' in pairs, or by using the character '\', is referred to as *quotation* (1.7).

redirection — The routing of input or output from or to a file is known as *redirection* of input or output (1.3).

rehash — The *rehash* command tells the shell to rebuild its internal table of which commands are found in which directories in your *path*. This is necessary when a new program is installed in one of these directories (2.8).

relative pathname — A *pathname* which does not begin with a '/' is called a *relative pathname* since it is interpreted *relative* to the current *working directory*. The first *component* of such a *pathname* refers to some file or directory in the *working directory*, and subsequent *components* between '/' characters refer to directories below the *working directory*. *Pathnames* that are not *relative* are called *absolute pathnames* (1.6).

repeat — The *repeat* command iterates another command a specified number of times.

root — The directory that is at the top of the entire directory structure is called the *root* directory since it is the 'root' of the entire tree structure of directories. The name used in *pathnames* to indicate the *root* is '/'. *Pathnames* starting with '/' are said to be *absolute* since they start at the *root* directory. *Root* is also used as the part of a *pathname* that is left after removing the *extension*. See *filename* for a further explanation (1.6).

RUBOUT — The RUBOUT or DELETE key is often used to erase the previously typed character; some users prefer the BACKSPACE for this purpose. On older versions of UNIX this key served as the INTR character.

scratch file — Files whose names begin with a '#' are referred to as *scratch files*, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight (1.3).

script — Sequences of shell commands placed in a file are called shell command *scripts*. It is often possible to perform simple tasks using these *scripts* without writing a program in a language such as C, by using the shell to selectively run other programs (3.3, 3.10).

set — The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (2.1).

setenv — Variables in the environment 'environ (5)' can be changed by using the *setenv* builtin command (2.8). The *printenv* command can be used to print the value of the variables in the environment.

shell — A *shell* is a command language interpreter. It is possible to write and run your own *shell*, as *shells* are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular *shell*, called *csh*.

shell script — See *script* (3.3, 3.10).

signal — A *signal* in UNIX is a short message that is sent to a running program which causes something to happen to that process. *Signals* are sent either by typing

special *control* characters on the keyboard or by using the *kill* or *stop* commands (1.8, 2.6).

sort — The *sort* program sorts a sequence of lines in ways that can be controlled by argument *flags* (1.5).

source — The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.8).

special character — See *metacharacters* and the appendix to this manual.

standard — We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).

status — A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero *status* to indicate that some abnormal event has occurred. The shell variable *status* is set to the *status* returned by the last command. It is most useful in shell commmand scripts (3.6).

stop — The *stop* command causes a *background* job to become *suspended* (2.6).

string — A sequential group of characters taken together is called a *string*. *Strings* can contain any printable characters (2.2).

stty — The *stty* program changes certain parameters inside UNIX which determine how your terminal is handled. See 'stty (1)' for a complete description (2.6).

substitution — The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history *substitution* keyed by the metacharacter '!' and variable *substitution* indicated by '$'. We also refer to *substitutions* as *expansions* (3.4).

suspended — A job becomes *suspended* after a STOP signal is sent to it, either by typing a *control-z* at the terminal (for *foreground* jobs) or by using the *stop* command (for *background* jobs). When *suspended*, a job temporarily stops running until it is restarted by either the *fg* or *bg* command (2.6).

switch — The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7).

termination — When a command which is being executed finishes we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an *end-of-file* from their *standard input*. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified jobs (2.6).

then — The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6).

time — The *time* command can be used to measure the amount of CPU and real time consumed by a specified command as well as the amount of disk i/o, memory utilized, and number of page faults and swaps taken by the command (2.1, 2.8).

tset — The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).

tty — The word *tty* is a historical abbreviation for 'teletype' which is frequently used in UNIX to indicate the *port* to which a given terminal is connected. The *tty* command will print the name of the *tty* or *port* to which your terminal is presently connected.

unalias — The *unalias* command removes aliases (2.8).

UNIX — UNIX is an operating system on which *csh* runs. UNIX provides facilities which allow *csh* to invoke other programs such as editors and text formatters which you may wish to use.

unset — The *unset* command removes the definitions of shell variables (2.2, 2.8).

variable expansion — See *variables* and *expansion* (2.2, 3.4).

variables — *Variables* in *csh* hold one or more strings as value. The most common use of *variables* is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. *Variables* such as *argv* are also used in writing shell programs (shell command scripts) (2.2).

verbose — The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shell's *-v* command line option (3.10).

wc — The *wc* program calculates the number of characters, words, and lines in the files whose names are given as arguments (2.6).

while — The *while* builtin control construct is used in shell command scripts (3.7).

word — A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '–', '.' nor '/' form *words* all by themselves even if they are not surrounded by blanks. Any sequence of characters may be made into a *word* by surrounding it with '' characters except for the characters '' and '!' which require special treatment (1.1). This process of placing special characters in *words* without their special meaning is called *quoting*.

working directory — At any given time you are in one particular directory, called your *working directory*. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change *working directories* using *chdir*.

write — The *write* command is an obsolete way of communicating with other users who are logged in to UNIX (you have to take turns typing). If you are both using display terminals, use *talk*(1), which is much more pleasant.

An Introduction to the UNIX Shell

S. R. Bourne

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

The *shell*‡ is a command programming language that provides an interface to the UNIX† operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while, if then else, case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through 'pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

1.0 Introduction

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section; see, for example, "UNIX for beginners". unix beginn kernigh 1978 Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form "see *pipe* (2)" are to a section of the UNIX manual. seventh 1978 ritchie thompson

1.1 Simple commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument *-l* tells *ls* to print status information,

‡ This paper describes sh(1). If it's the c shell (csh) you're interested in, a good place to begin is William Joy's paper "An Introduction to the C shell" (USD:4).

† UNIX is a trademark of AT&T Bell Laboratories.

size and the creation date for each file.

1.2 Background commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing **&** is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps* command.

1.3 Input output redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
ls -l >file
```

The notation *>file* is interpreted by the shell and is not passed as an argument to *ls*. If *file* does not exist then the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls*. Output may be appended to a file using the notation

```
ls -l >>file
```

In this case *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

```
wc <file
```

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words and lines found. If only the number of lines is required then

```
wc -l <file
```

could be used.

1.4 Pipelines and filters

The standard output of one command may be connected to the standard input of another by writing the 'pipe' operator, indicated by |, as in,

```
ls -l | wc
```

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe (see *pipe* (2)) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep*, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from *ls* that contain the string *old*. Another useful filter is *sort*. For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

1.5 File name generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names in the current directory that end in *.c*. The character * is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*.

```
/usr/fred/test/?
```

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of **/usr/fred**. (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of **/usr/fred**.

There is one exception to the general rules given for patterns. The character '.' at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with '.'.

```
echo .*
```

will echo all those file names that begin with '.'. This avoids inadvertent matching of the names '.' and '..' which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files '.' and '..'.)

1.6 Quoting

Characters that have a special meaning to the shell, such as < > * ? | &, are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a \

is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single **?**, and

```
echo \\
```

will echo a single ****. To allow long strings to be continued over more than one line the sequence **\newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4.

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is '$ '. It may be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. If a newline is typed and further input is needed then the shell will issue the prompt '> '. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

```
PS2=more
```

1.8 The shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile** then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- **ls**
  Print the names of files in the current directory.
- **ls >file**
  Put the output from *ls* into *file*.
- **ls | wc –l**
  Print the number of files in the current directory.
- **ls | grep old**
  Print those file names containing the string *old*.
- **ls | grep old | wc –l**
  Print the number of files whose name contains the string *old*.

- **cc pgm.c &**
  Run *cc* in the background.

2.0 Shell procedures

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1, $2, ...** For example, if the file *wg* contains

```
who ! grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who ! grep fred
```

UNIX files have three independent attributes, *read, write* and *execute*. The UNIX command *chmod* (1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special shell parameter **$*** is used to substitute for all positional parameters except **$0**. A typical use of this is to provide some default arguments, as in,

```
nroff –T450 –ms $*
```

which simply prepends some arguments to those already given.

2.1 Control flow - for

A frequent use of shell procedures is to loop through the arguments (**$1, $2, ...**) executing commands once for each argument. An example of such a procedure is *tel* that searches the file **/usr/lib/telnos** that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do grep $i /usr/lib/telnos; done
```

The command

```
tel fred
```

prints those lines in **/usr/lib/telnos** that contain the string *fred*.

```
tel fred bert
```

prints those lines containing *fred* followed by those for *bert*.

The **for** loop notation is recognized by the shell and has the general form

for *name* **in** *w1 w2* ...
do *command-list*
done

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. *name* is a shell variable that is set to the words *w1 w2* ... in turn each time the *command-list* following **do** is executed. If **in** *w1 w2* ... is omitted then the loop is executed once for each positional parameter; that is, **in** $* is assumed.

Another example of the use of the **for** loop is the *create* command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done.**

2.2 Control flow - case

A multiple way branch is provided for by the **case** notation. For example,

```
case $# in
    1)    cat >>$1 ;;
    2)    cat >>$2 <$1 ;;
    *)    echo 'usage: append [ from ] to' ;;
esac
```

is an *append* command. When called with one argument as

```
append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the **case** command is

case *word* **in**
    pattern) *command-list*;;
    ...
esac

The shell attempts to match *word* with each *pattern,* in the order in which the patterns appear. If a match is found the associated *command-list* is executed and execution of the **case** is complete. Since * is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the

commands following the second * will never be executed.

```
case $# in
    *) ... ;;
    *) ... ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```
for i
do case $i in
   -[ocs]) ... ;;
   -*)     echo 'unknown flag $i' ;;
   *.c)    /lib/c0 $i ... ;;
   *) echo 'unexpected argument $i' ;;
   esac
done
```

To allow the same commands to be associated with more than one pattern the **case** command provides for alternative patterns separated by a | . For example,

```
case $i in
    -x|-y)      ...
esac
```

is equivalent to

```
case $i in
    -[xy]) ...
esac
```

The usual quoting conventions apply so that

```
case $i in
    \?)   ...
```

will match the character **?**.

2.3 Here documents

The shell procedure *tel* in section 2.1 uses the file **/usr/lib/telnos** to supply the data for *grep*. An alternative is to include this data within the shell procedure as a *here* document, as in,

```
for i
do grep $i <<!
   ...
   fred mh0123
   bert mh0789
   ...
!
done
```

In this example the shell takes the lines between **<<!** and **!** as the standard input for *grep*. The string **!** is arbitrary, the document being terminated by a line that consists of the string following **<<**.

Parameters are substituted in the document before it is made available to *grep* as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character $ as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

(This version of *edg* is equivalent to the first except that *ed* will print a **?** if there are no occurrences of the string **$1.**) Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document this latter form is more efficient.

2.4 Shell variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables **user, box** and **acct.** A variable may be set to the null string by saying, for example,

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo *fred.*

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

will move the file *pgm* from the current directory to the directory **/usr/fred/bin.** A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of *ps* to the file **/tmp/psa,** whereas,

```
ps a >$tmpa
```

would cause the value of the variable **tmpa** to be substituted.

Except for $? the following are set initially by the shell. $? is set after executing each command.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). Used, for example, in the *append* command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$– The current shell flags, such as **–x** and **–v**.

Some variables have a special meaning to the shell and should be avoided for general use.

$MAIL When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file **.profile,** in the user's login directory. For example,

```
MAIL=/usr/spool/mail/fred
```

$HOME The default argument for the *cd* command. The current directory is used to resolve file name references that do not begin with a **/**, and is changed using the *cd* command. For example,

```
cd /usr/fred/bin
```

makes the current directory **/usr/fred/bin.**

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command *cd* with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the the user's login profile.

$PATH A list of directories that contain commands (the *search path*). Each time a com-

mand is executed by the shell a list of directories is searched for an executable file. If **$PATH** is not set then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise **$PATH** consists of directory names separated by **:**. For example,

PATH=:/usr/fred/bin:/bin:/usr/bin

specifies that the current directory (the null string before the first **:**), **/usr/fred/bin, /bin** and **/usr/bin** are to be searched in that order. In this way individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a / then this directory search is not used; a single attempt is made to execute the command.

$PS1 The primary shell prompt string, by default, '$ '.

$PS2 The shell prompt when further input is needed, by default, '> '.

$IFS The set of characters used by *blank interpretation* (see section 3.4).

2.5 The test command

The *test* command, although not part of the shell, is intended for use by shell programs. For example,

```
test –f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here, see *test* (1) for a complete specification.

```
test s       true if the argument s is not the null string
test –f file true if file exists
test –r file true if file is readable
test –w file true if file is writable
test –d file true if file is a directory
```

2.6 Control flow - while

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

while *command-list*$_1$
do *command-list*$_2$
done

The value tested by the **while** command is the exit status of the last simple command following **while.** Each time round the loop *command-list*$_1$ is executed; if a zero exit status is returned then *command-list*$_2$ is executed; otherwise, the loop terminates. For example,

```
while test $1
do ...
   shift
done
```

is equivalent to

```
for i
do ...
done
```

shift is a shell command that renames the positional parameters **$2, $3, ...** as **$1, $2, ...** and loses **$1.**

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop the termination condition is reversed. For example,

```
until test –f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control flow - if

Also available is a general conditional branch of the form,

```
if command-list
then command-list
else command-list
fi
```

that tests the value returned by the last simple command following **if.**

The **if** command may be used in conjunction with the *test* command to test for the existence of a file as in

```
if test –f file
then process file
else do something else
fi
```

An example of the use of **if, case** and **for** constructions is given in section 2.10.

A multiple test **if** command of the form

```
if ...
then ...
else if ...
     then ...
     else if ...
          ...
          fi
     fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

The following example is the *touch* command which changes the 'last modified' time for a list of files. The command may be used in conjunction with *make* (1) to force recompilation of a list of files.

```
flag=
for i
do case $i in
   -c)      flag=N ;;
   *) if test -f $i
      then  ln $i junk$$; rm junk$$
      elif test $flag
      then  echo file \'$i\' does not exist
      else  >$i
      fi
   esac
done
```

The **-c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **-c** argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```
if command1
then  command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes *command2* only if *command1* fails. In each case the value returned is that of the last simple command executed.

2.8 Command grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

In the first *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory **x** without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory **x.**

2.9 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```

(**v** for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the **-n** flag which prevents execution of subsequent commands. (Note that saying *set -n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

```
set -
```

and the current setting of the shell flags is available as **$-**.

2.10 The man command

The following is the *man* command which is used to diplay sections of the UNIX manual on your terminal. It is called, for example, as

```
man sh
man -t ed
man 2 fork
```

In the first the manual section for *sh* is displayed.. Since no section is specified, section **1** is used. The second example will typeset (**-t** option) the manual section for *ed*. The last prints the *fork* manual page from section 2, which covers system calls.

```
cd /usr/man

: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1

for i
do case $i in
   [1-9]*) s=$i ;;
   -t)     N=t ;;
   -n)     N=n ;;
   -*)     echo unknown flag \'$i\' ;;
   *) if test -f man$s/$i.$s
      then ${N}roff man0/${N}aa man$s/$i.$s
      else : 'look through all manual sections'
           found=no
           for j in 1 2 3 4 5 6 7 8 9
           do if test -f man$j/$i.$j
              then man $j $i
                   found=yes
              fi
           done
           case $found in
                no) echo '$i: manual page not found'
           esac
      fi
   esac
done
```

Figure 1. A version of the man command

3.0 Keyword parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

```
user=fred command
```

will execute *command* with **user** set to *fred*. The **-k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain they are available as positional parameters **$1, $2, ...**

The *set* command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set **$1** to the first file name in the current directory, **$2** to the next, and so on. Note that the first argument, -, ensures correct treatment when the first file name begins with a -.

3.1 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables **user** and **box** for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.2 Parameter substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable **d** is not set

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.}
```

which will echo the value of the variable **d** if it is set and '.' otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d-'*'}
```

will echo * if the variable **d** is not set. Similarly

```
echo ${d-$1}
```

will echo the value of **d** if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if **d** were not previously set then it will be set to the string '.'. (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct** or **bin** are not set then the shell will abandon execution of the procedure.

3.3 Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is **/usr/fred/bin** then the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (`...`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a \. For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

```
case $A in
    ...
    *.c)    B=`basename $A .c`
    ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- **for i in `ls -t`; do ...**
  The variable **i** is set to the names of files in time order, most recent first.
- **set `date`; echo $6 $2 $3, $4**
  will print, e.g., *1977 Nov 1, 23:59:59*

3.4 Evaluation and quoting

The shell is a macro processor that provides parameter substitution, command substitution and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in appendix A. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. **$user**
- command substitution, e.g. **`pwd`**

  Only one evaluation occurs so that if, for example, the value of the variable **X** is the string $*y* then

```
echo $X
```

will echo $*y*.

- blank interpretation

  Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string $IFS. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

  ```
  echo ''
  ```

  will pass on the null string as the first argument to *echo*, whereas

  ```
  echo $null
  ```

  will call *echo* with no arguments if the variable **null** is not set or set to the null string.

- file name generation

  Each word is then scanned for the file pattern characters *, ? and [...] and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and '...' a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

will pass the value of the variable **x** as a single argument to *echo*. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted.

```
echo "$@"
```

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

	metacharacter					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t terminator
y interpreted
n not interpreted

Figure 2. Quoting mechanisms

In cases where more than one evaluation of a string is required the built-in command *eval* may be used. For example, if the variable **X** has the value $*y*, and if **y** has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

In general the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who | grep'
$wg fred
```

is equivalent to

```
who | grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as | , **following substitution.**

3.5 Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *gtty* (2)). A shell invoked with the **–i** flag is also interactive.

Execution of a command (see also 3.7) may fail for any of the following reasons.

- Input output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See Figure 2 below for a complete list of UNIX signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- Syntax errors. e.g., if ... then ... done
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as *cd*.

The shell flag **–e** causes the shell to terminate if any error is detected.

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from *kill* (1))

Figure 3. UNIX signals†

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The *exit* is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file **junk$$**.

† Additional signals have been added in Berkeley Unix. See sigvec(2) or signal(3C) for an up-to-date list.

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do case $i in
   -c)      flag=N ;;
   *) if test -f $i
      then  ln $i junk$$; rm junk$$
      elif test $flag
      then  echo file \'$i\' does not exist
      else  >$i
      fi
   esac
done
```

Figure 4. The touch command

The *trap* command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in UNIX it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command.

```
trap '' 1 2 3 15
```

which causes *hangup, interrupt, quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The procedure *scan* (Figure 5) is an example of the use of *trap* where there is no exit in the trap command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

```
d=`pwd`
for i in *
do if test -d $d/$i
   then cd $d/$i
        while echo "$i:"
              trap exit 2
              read x
        do trap : 2; eval $x; done
   fi
done
```

Figure 5. The scan command

read x is a built-in command that reads one line from the standard input and places the result in the variable **x**. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap '' 1 2 3 15
exec $*
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is ***.c**. Input output specifications are evaluated left to right as they appear in the command.

> *word*	The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
>> *word*	The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
< *word*	The standard input (file descriptor 0) is taken from the file *word*.
<< *word*	The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted then parameter and command substitution occur and \ is used to quote the characters \ $ ` and the first character of *word*. In the latter case **\newline** is ignored (c.f. quoted strings).
>& *digit*	The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
<& *digit*	The standard input is duplicated from file descriptor *digit*.
<&-	The standard input is closed.
>&-	The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty file

/dev/null. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason the UNIX convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

3.8 Invoking the shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file **.profile.**

–c *string*
: If the **–c** flag is present then commands are read from *string*.

–s
: If the **–s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

–i
: If the **–i** flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptable). In all cases QUIT is ignored by the shell.

Acknowledgements

The design of the shell is based in part on the original UNIX shell unix command language thompson and the PWB/UNIX shell, pwb shell mashey unix some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System cambridge multiple access system hartley and of CTSS. ctss

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center and to Joe Maranzano for their comments on drafts of this document.

$LIST$

Appendix A - Grammar

item: *word*
input-output
name = *value*

simple-command: *item*
simple-command item

command: *simple-command*
(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* ... **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* ... **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: *command*
pipeline | *command*

andor: *pipeline*
andor && *pipeline*
andor || *pipeline*

command-list: *andor*
command-list ;
command-list &
command-list ; *andor*
command-list & *andor*

input-output: > *file*
< *file*
>> *word*
<< *word*

file: *word*
& *digit*
& -

case-part: *pattern*) *command-list* ;;

pattern: *word*
pattern | *word*

else-part: **elif** *command-list* **then** *command-list else-part*
else *command-list*
empty

empty:

word: a sequence of non-blank characters

name: a sequence of letters, digits or underscores starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

Appendix B - Meta-characters and Reserved Words

a) syntactic

|	pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
>>	output append

b) patterns

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

c) substitution

${...}	substitute shell variable
`...`	substitute command output

d) quoting

\	quote the next character
'...'	quote the enclosed characters except for '
"..."	quote the enclosed characters except for $ ` \ "

e) reserved words

if then else elif fi
case in esac
for while until do done
{ }

MAIL REFERENCE MANUAL

Kurt Shoens

Revised by

Craig Leres

Version 5.2

October 1, 1991

1. Introduction

Mail provides a simple and friendly environment for sending and receiving mail. It divides incoming mail into its constituent messages and allows the user to deal with them in any order. In addition, it provides a set of *ed*-like commands for manipulating messages and sending mail. *Mail* offers the user simple editing capabilities to ease the composition of outgoing messages, as well as providing the ability to define and send to names which address groups of users. Finally, *Mail* is able to send and receive messages across such networks as the ARPANET, UUCP, and Berkeley network.

This document describes how to use the *Mail* program to send and receive messages. The reader is not assumed to be familiar with other message handling systems, but should be familiar with the UNIX[1] shell, the text editor, and some of the common UNIX commands. "The UNIX Programmer's Manual," "An Introduction to Csh," and "Text Editing with Ex and Vi" can be consulted for more information on these topics.

Here is how messages are handled: the mail system accepts incoming *messages* for you from other people and collects them in a file, called your *system mailbox.* When you login, the system notifies you if there are any messages waiting in your system mailbox. If you are a *csh* user, you will be notified when new mail arrives if you inform the shell of the location of your mailbox. On version 7 systems, your system mailbox is located in the directory /usr/spool/mail in a file with your login name. If your login name is "sam," then you can make *csh* notify you of new mail by including the following line in your .cshrc file:

```
set mail=/usr/spool/mail/sam
```

When you read your mail using *Mail,* it reads your system mailbox and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, or save these messages. Each message is marked with its author and the date they sent it.

[1] UNIX is a trademark of Bell Laboratories.

2. Common usage

The *Mail* command has two distinct usages, according to whether one wants to send or receive mail. Sending mail is simple: to send a message to a user whose login name is, say, "root," use the shell command:

% Mail root

then type your message. When you reach the end of the message, type an EOT (control–d) at the beginning of a line, which will cause *Mail* to echo "EOT" and return you to the Shell. When the user you sent mail to next logs in, he will receive the message:

You have mail.

to alert him to the existence of your message.

If, while you are composing the message you decide that you do not wish to send it after all, you can abort the letter with a RUBOUT. Typing a single RUBOUT causes *Mail* to print

(Interrupt -- one more to kill letter)

Typing a second RUBOUT causes *Mail* to save your partial letter on the file "dead.letter" in your home directory and abort the letter. Once you have sent mail to someone, there is no way to undo the act, so be careful.

The message your recipient reads will consist of the message you typed, preceded by a line telling who sent the message (your login name) and the date and time it was sent.

If you want to send the same message to several other people, you can list their login names on the command line. Thus,

% Mail sam bob john
Tuition fees are due next Friday. Don't forget!!
<Control–d>
EOT
%

will send the reminder to sam, bob, and john.

If, when you log in, you see the message,

You have mail.

you can read the mail by typing simply:

% Mail

Mail will respond by typing its version number and date and then listing the messages you have waiting. Then it will type a prompt and await your command. The messages are assigned numbers starting with 1 — you refer to the messages with these numbers. *Mail* keeps tack of which messages are *new* (have been sent since you last read your mail) and *read* (have been read by you). New messages have an **N** next to them in the header listing and old, but unread messages have a **U** next to them. *Mail* keeps track of new/old and read/unread messages by putting a header field called "Status" into your messages.

To look at a specific message, use the **type** command, which may be abbreviated to simply **t**. For example, if you had the following messages:

N 1 root Wed Sep 21 09:21 "Tuition fees"
N 2 sam Tue Sep 20 22:55

you could examine the first message by giving the command:

type 1

which might cause *Mail* to respond with, for example:

Message 1:

```
From root  Wed Sep 21 09:21:45 1978
Subject: Tuition fees
Status: R

Tuition fees are due next Wednesday.  Don't forget!!
```

Many *Mail* commands that operate on messages take a message number as an argument like the **type** command. For these commands, there is a notion of a current message. When you enter the *Mail* program, the current message is initially the first one. Thus, you can often omit the message number and use, for example,

```
t
```

to type the current message. As a further shorthand, you can type a message by simply giving its message number. Hence,

```
1
```

would type the first message.

Frequently, it is useful to read the messages in your mailbox in order, one after another. You can read the next message in *Mail* by simply typing a newline. As a special case, you can type a newline as your first command to *Mail* to type the first message.

If, after typing a message, you wish to immediately send a reply, you can do so with the **reply** command. **Reply**, like **type**, takes a message number as an argument. *Mail* then begins a message addressed to the user who sent you the message. You may then type in your letter in reply, followed by a <control-d> at the beginning of a line, as before. *Mail* will type EOT, then type the ampersand prompt to indicate its readiness to accept another command. In our example, if, after typing the first message, you wished to reply to it, you might give the command:

```
reply
```

Mail responds by typing:

```
To: root
Subject: Re: Tuition fees
```

and waiting for you to enter your letter. You are now in the message collection mode described at the beginning of this section and *Mail* will gather up your message up to a control–d. Note that it copies the subject header from the original message. This is useful in that correspondence about a particular matter will tend to retain the same subject heading, making it easy to recognize. If there are other header fields in the message, the information found will also be used. For example, if the letter had a "To:" header listing several recipients, *Mail* would arrange to send your replay to the same people as well. Similarly, if the original message contained a "Cc:" (carbon copies to) field, *Mail* would send your reply to *those* users, too. *Mail* is careful, though, not too send the message to *you*, even if you appear in the "To:" or "Cc:" field, unless you ask to be included explicitly. See section 4 for more details.

After typing in your letter, the dialog with *Mail* might look like the following:

```
reply
To: root
Subject: Tuition fees

Thanks for the reminder
EOT
&
```

The **reply** command is especially useful for sustaining extended conversations over the message system, with other "listening" users receiving copies of the conversation. The **reply**

command can be abbreviated to **r**.

Sometimes you will receive a message that has been sent to several people and wish to reply *only* to the person who sent it. **Reply** with a capital **R** replies to a message, but sends a copy to the sender only.

If you wish, while reading your mail, to send a message to someone, but not as a reply to one of your messages, you can send the message directly with the **mail** command, which takes as arguments the names of the recipients you wish to send to. For example, to send a message to "frank," you would do:

```
mail frank
This is to confirm our meeting next Friday at 4.
EOT
&
```

The **mail** command can be abbreviated to **m**.

Normally, each message you receive is saved in the file *mbox* in your login directory at the time you leave *Mail*. Often, however, you will not want to save a particular message you have received because it is only of passing interest. To avoid saving a message in *mbox* you can delete it using the **delete** command. In our example,

```
delete 1
```

will prevent *Mail* from saving message 1 (from root) in *mbox*. In addition to not saving deleted messages, *Mail* will not let you type them, either. The effect is to make the message disappear altogether, along with its number. The **delete** command can be abbreviated to simply **d**.

Many features of *Mail* can be tailored to your liking with the **set** command. The **set** command has two forms, depending on whether you are setting a *binary* option or a *valued* option. Binary options are either on or off. For example, the "ask" option informs *Mail* that each time you send a message, you want it to prompt you for a subject header, to be included in the message. To set the "ask" option, you would type

```
set ask
```

Another useful *Mail* option is "hold." Unless told otherwise, *Mail* moves the messages from your system mailbox to the file *mbox* in your home directory when you leave *Mail*. If you want *Mail* to keep your letters in the system mailbox instead, you can set the "hold" option.

Valued options are values which *Mail* uses to adapt to your tastes. For example, the "SHELL" option tells *Mail* which shell you like to use, and is specified by

```
set SHELL=/bin/csh
```

for example. Note that no spaces are allowed in "SHELL=/bin/csh." A complete list of the *Mail* options appears in section 5.

Another important valued option is "crt." If you use a fast video terminal, you will find that when you print long messages, they fly by too quickly for you to read them. With the "crt" option, you can make *Mail* print any message larger than a given number of lines by sending it through the paging program *more*. For example, most CRT users with 24-line screens should do:

```
set crt=24
```

to paginate messages that will not fit on their screens. *More* prints a screenful of information, then types --MORE--. Type a space to see the next screenful.

Another adaptation to user needs that *Mail* provides is that of *aliases*. An alias is simply a name which stands for one or more real user names. *Mail* sent to an alias is really sent to the list of real users associated with it. For example, an alias can be defined for the members of a project, so that you can send mail to the whole project by sending mail to just a single name. The **alias** command in *Mail* defines an alias. Suppose that the users in a project are named Sam,

Sally, Steve, and Susan. To define an alias called "project" for them, you would use the *Mail* command:

alias project sam sally steve susan

The **alias** command can also be used to provide a convenient name for someone whose user name is inconvenient. For example, if a user named "Bob Anderson" had the login name "anderson,"" you might want to use:

alias bob anderson

so that you could send mail to the shorter name, "bob."

While the **alias** and **set** commands allow you to customize *Mail*, they have the drawback that they must be retyped each time you enter *Mail*. To make them more convenient to use, *Mail* always looks for two files when it is invoked. It first reads a system wide file "/usr/lib/Mail.rc," then a user specific file, ".mailrc," which is found in the user's home directory. The system wide file is maintained by the system administrator and contains **set** commands that are applicable to all users of the system. The ".mailrc" file is usually used by each user to set options the way he likes and define individual aliases. For example, my .mailrc file looks like this:

set ask nosave SHELL=/bin/csh

As you can see, it is possible to set many options in the same **set** command. The "nosave" option is described in section 5.

Mail aliasing is implemented at the system-wide level by the mail delivery system *sendmail*. These aliases are stored in the file /usr/lib/aliases and are accessible to all users of the system. The lines in /usr/lib/aliases are of the form:

alias: $name_1$, $name_2$, $name_3$

where *alias* is the mailing list name and the $name_i$ are the members of the list. Long lists can be continued onto the next line by starting the next line with a space or tab. Remember that you must execute the shell command *newaliases* after editing /usr/lib/aliases since the delivery system uses an indexed file created by *newaliases*.

We have seen that *Mail* can be invoked with command line arguments which are people to send the message to, or with no arguments to read mail. Specifying the **–f** flag on the command line causes *Mail* to read messages from a file other than your system mailbox. For example, if you have a collection of messages in the file "letters" you can use *Mail* to read them with:

% Mail –f letters

You can use all the *Mail* commands described in this document to examine, modify, or delete messages from your "letters" file, which will be rewritten when you leave *Mail* with the **quit** command described below.

Since mail that you read is saved in the file *mbox* in your home directory by default, you can read *mbox* in your home directory by using simply

% Mail –f

Normally, messages that you examine using the **type** command are saved in the file "mbox" in your home directory if you leave *Mail* with the **quit** command described below. If you wish to retain a message in your system mailbox you can use the **preserve** command to tell *Mail* to leave it there. The **preserve** command accepts a list of message numbers, just like **type** and may be abbreviated to **pre**.

Messages in your system mailbox that you do not examine are normally retained in your system mailbox automatically. If you wish to have such a message saved in *mbox* without reading it, you may use the **mbox** command to have them so saved. For example,

mbox 2

in our example would cause the second message (from sam) to be saved in *mbox* when the **quit** command is executed. **Mbox** is also the way to direct messages to your *mbox* file if you have set the "hold" option described above. **Mbox** can be abbreviated to **mb**.

When you have perused all the messages of interest, you can leave *Mail* with the **quit** command, which saves the messages you have typed but not deleted in the file *mbox* in your login directory. Deleted messages are discarded irretrievably, and messages left untouched are preserved in your system mailbox so that you will see them the next time you type:

% Mail

The **quit** command can be abbreviated to simply **q**.

If you wish for some reason to leave *Mail* quickly without altering either your system mailbox or *mbox*, you can type the **x** command (short for **exit**), which will immediately return you to the Shell without changing anything.

If, instead, you want to execute a Shell command without leaving *Mail*, you can type the command preceded by an exclamation point, just as in the text editor. Thus, for instance:

!date

will print the current date without leaving *Mail*.

Finally, the **help** command is available to print out a brief summary of the *Mail* commands, using only the single character command abbreviations.

3. Maintaining folders

Mail includes a simple facility for maintaining groups of messages together in folders. This section describes this facility.

To use the folder facility, you must tell *Mail* where you wish to keep your folders. Each folder of messages will be a single file. For convenience, all of your folders are kept in a single directory of your choosing. To tell *Mail* where your folder directory is, put a line of the form

set folder=letters

in your *.mailrc* file. If, as in the example above, your folder directory does not begin with a '/,' *Mail* will assume that your folder directory is to be found starting from your home directory. Thus, if your home directory is **/usr/person** the above example told *Mail* to find your folder directory in **/usr/person/letters**.

Anywhere a file name is expected, you can use a folder name, preceded with '+.' For example, to put a message into a folder with the **save** command, you can use:

save +classwork

to save the current message in the *classwork* folder. If the *classwork* folder does not yet exist, it will be created. Note that messages which are saved with the **save** command are automatically removed from your system mailbox.

In order to make a copy of a message in a folder without causing that message to be removed from your system mailbox, use the **copy** command, which is identical in all other respects to the **save** command. For example,

copy +classwork

copies the current message into the *classwork* folder and leaves a copy in your system mailbox.

The **folder** command can be used to direct *Mail* to the contents of a different folder. For example,

folder +classwork

directs *Mail* to read the contents of the *classwork* folder. All of the commands that you can use on your system mailbox are also applicable to folders, including **type**, **delete**, and **reply**. To

inquire which folder you are currently editing, use simply:

```
folder
```

To list your current set of folders, use the **folders** command.

To start *Mail* reading one of your folders, you can use the **–f** option described in section 2. For example:

```
% Mail –f +classwork
```

will cause *Mail* to read your ***classwork*** folder without looking at your system mailbox.

4. More about sending mail

4.1. Tilde escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or do some other auxiliary function. *Mail* provides these capabilities through *tilde escapes*, which consist of a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed. For example, to print the text of the message so far, use:

~p

which will print a line of dashes, the recipients of your message, and the text of the message so far. Since *Mail* requires two consecutive RUBOUT's to abort a letter, you can use a single RUBOUT to abort the output of ~p or any other ~ escape without killing your letter.

If you are dissatisfied with the message as it stands, you can invoke the text editor on it using the escape

~e

which causes the message to be copied into a temporary file and an instance of the editor to be spawned. After modifying the message to your satisfaction, write it out and quit the editor. *Mail* will respond by typing

(continue)

after which you may continue typing text which will be appended to your message, or type <control-d> to end the message. A standard text editor is provided by *Mail*. You can override this default by setting the valued option "EDITOR" to something else. For example, you might prefer:

set EDITOR=/usr/ucb/ex

Many systems offer a screen editor as an alternative to the standard text editor, such as the *vi* editor from UC Berkeley. To use the screen, or *visual* editor, on your current message, you can use the escape,

~v

~v works like ~e, except that the screen editor is invoked instead. A default screen editor is defined by *Mail*. If it does not suit you, you can set the valued option "VISUAL" to the path name of a different editor.

It is often useful to be able to include the contents of some file in your message; the escape

~r filename

is provided for this purpose, and causes the named file to be appended to your current message. *Mail* complains if the file doesn't exist or can't be read. If the read is successful, the number of lines and characters appended to your message is printed, after which you may continue appending text. The filename may contain shell metacharacters like * and ? which are expanded according to the conventions of your shell.

As a special case of ~r, the escape

~d

reads in the file "dead.letter" in your home directory. This is often useful since *Mail* copies the text of your message there when you abort a message with RUBOUT.

To save the current text of your message on a file you may use the

~w filename

escape. *Mail* will print out the number of lines and characters written to the file, after which you may continue appending text to your message. Shell metacharacters may be used in the filename,

as in ~r and are expanded with the conventions of your shell.

If you are sending mail from within *Mail's* command mode you can read a message sent to you into the message you are constructing with the escape:

~m 4

which will read message 4 into the current message, shifted right by one tab stop. You can name any non-deleted message, or list of messages. Messages can also be forwarded without shifting by a tab stop with ~f. This is the usual way to forward a message.

If, in the process of composing a message, you decide to add additional people to the list of message recipients, you can do so with the escape

~t name1 name2 ...

You may name as few or many additional recipients as you wish. Note that the users originally on the recipient list will still receive the message; you cannot remove someone from the recipient list with ~t.

If you wish, you can associate a subject with your message by using the escape

~s Arbitrary string of text

which replaces any previous subject with "Arbitrary string of text." The subject, if given, is sent near the top of the message prefixed with "Subject:" You can see what the message will look like by using ~p.

For political reasons, one occasionally prefers to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape

~c name1 name2 ...

adds the named people to the "Cc:" list, similar to ~t. Again, you can execute ~p to see what the message will look like.

The recipients of the message together constitute the "To:" field, the subject the "Subject:" field, and the carbon copies the "Cc:" field. If you wish to edit these in ways impossible with the ~t, ~s, and ~c escapes, you can use the escape

~h

which prints "To:" followed by the current list of recipients and leaves the cursor (or printhead) at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use your erase character to erase back into the list of recipients, or your kill character to erase them altogether. Thus, for example, if your erase and kill characters are the standard (on printing terminals) # and @ symbols,

~h
To: root kurt####bill

would change the initial recipients "root kurt" to "root bill." When you type a newline, *Mail* advances to the "Subject:" field, where the same rules apply. Another newline brings you to the "Cc:" field, which may be edited in the same fashion. Another newline leaves you appending text to the end of your message. You can use ~p to print the current text of the header fields and the body of the message.

To effect a temporary escape to the shell, the escape

~!command

is used, which executes *command* and returns you to mailing mode without altering the text of your message. If you wish, instead, to filter the body of your message through a shell command, then you can use

~|command

which pipes your message through the command and uses the output as the new text of your

message. If the command produces no output, *Mail* assumes that something is amiss and retains the old version of your message. A frequently-used filter is the command *fmt*, designed to format outgoing mail.

To effect a temporary escape to *Mail* command mode instead, you can use the

~:*Mail command*

escape. This is especially useful for retyping the message you are replying to, using, for example:

~:t

It is also useful for setting options and modifying aliases.

If you wish (for some reason) to send a message that contains a line beginning with a tilde, you must double it. Thus, for example,

~~This line begins with a tilde.

sends the line

~This line begins with a tilde.

Finally, the escape

~?

prints out a brief summary of the available tilde escapes.

On some terminals (particularly ones with no lower case) tilde's are difficult to type. *Mail* allows you to change the escape character with the "escape" option. For example, I set

set escape=]

and use a right bracket instead of a tilde. If I ever need to send a line beginning with right bracket, I double it, just as for ~. Changing the escape character removes the special meaning of ~.

4.2. Network access

This section describes how to send mail to people on other machines. Recall that sending to a plain login name sends mail to that person on your machine. If your machine is directly (or sometimes, even, indirectly) connected to the Arpanet, you can send messages to people on the Arpanet using a name of the form

name@host.domain

where *name* is the login name of the person you're trying to reach, *host* is the name of the machine on the Arpanet, and *domain* is the higher-level scope within which the hostname is known, e.g. EDU (for educational institutions), COM (for commercial entities), GOV (for governmental agencies), ARPA for many other things, BITNET or CSNET for those networks.

If your recipient logs in on a machine connected to yours by UUCP (the Bell Laboratories supplied network that communicates over telephone lines), sending mail can be a bit more complicated. You must know the list of machines through which your message must travel to arrive at his site. So, if his machine is directly connected to yours, you can send mail to him using the syntax:

host!name

where, again, *host* is the name of the machine and *name* is the login name. If your message must go through an intermediary machine first, you must use the syntax:

intermediary!host!name

and so on. It is actually a feature of UUCP that the map of all the systems in the network is not known anywhere (except where people decide to write it down for convenience). Talk to your system administrator about good ways to get places; the *uuname* command will tell you systems

whose names are recognized, but not which ones are frequently called or well-connected.

When you use the **reply** command to respond to a letter, there is a problem of figuring out the names of the users in the "To:" and "Cc:" lists *relative to the current machine.* If the original letter was sent to you by someone on the local machine, then this problem does not exist, but if the message came from a remote machine, the problem must be dealt with. *Mail* uses a heuristic to build the correct name for each user relative to the local machine. So, when you **reply** to remote mail, the names in the "To:" and "Cc:" lists may change somewhat.

4.3. Special recipients

As described previously, you can send mail to either user names or **alias** names. It is also possible to send messages directly to files or to programs, using special conventions. If a recipient name has a '/' in it or begins with a '+', it is assumed to be the path name of a file into which to send the message. If the file already exists, the message is appended to the end of the file. If you want to name a file in your current directory (ie, one for which a '/' would not usually be needed) you can precede the name with './' So, to send mail to the file "memo" in the current directory, you can give the command:

% Mail ./memo

If the name begins with a '+,' it is expanded into the full path name of the folder name in your folder directory. This ability to send mail to files can be used for a variety of purposes, such as maintaining a journal and keeping a record of mail sent to a certain group of users. The second example can be done automatically by including the full pathname of the record file in the **alias** command for the group. Using our previous **alias** example, you might give the command:

alias project sam sally steve susan /usr/project/mail_record

Then, all mail sent to "project" would be saved on the file "/usr/project/mail_record" as well as being sent to the members of the project. This file can be examined using *Mail -f.*

It is sometimes useful to send mail directly to a program, for example one might write a project billboard program and want to access it using *Mail.* To send messages to the billboard program, one can send mail to the special name '|billboard' for example. *Mail* treats recipient names that begin with a '|' as a program to send the mail to. An **alias** can be set up to reference a '|' prefaced name if desired. *Caveats*: the shell treats '|' specially, so it must be quoted on the command line. Also, the '| program' must be presented as a single argument to mail. The safest course is to surround the entire name with double quotes. This also applies to usage in the **alias** command. For example, if we wanted to alias 'rmsgs' to 'rmsgs –s' we would need to say:

alias rmsgs "| rmsgs -s"

5. Additional features

This section describes some additional commands useful for reading your mail, setting options, and handling lists of messages.

5.1. Message lists

Several *Mail* commands accept a list of messages as an argument. Along with **type** and **delete**, described in section 2, there is the **from** command, which prints the message headers associated with the message list passed to it. The **from** command is particularly useful in conjunction with some of the message list features described below.

A *message list* consists of a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters "↑" "." or "$" to specify the first relevant, current, or last relevant message, respectively. *Relevant* here means, for most commands "not deleted" and "deleted" for the **undelete** command.

A range of messages consists of two message numbers (of the form described in the previous paragraph) separated by a dash. Thus, to print the first four messages, use

```
type 1–4
```

and to print all the messages from the current message to the last message, use

```
type .–$
```

A *name* is a user name. The user names given in the message list are collected together and each message selected by other means is checked to make sure it was sent by one of the named users. If the message consists entirely of user names, then every message sent by one those users that is *relevant* (in the sense described earlier) is selected. Thus, to print every message sent to you by "root," do

```
type root
```

As a shorthand notation, you can specify simply "*" to get every *relevant* (same sense) message. Thus,

```
type *
```

prints all undeleted messages,

```
delete *
```

deletes all undeleted messages, and

```
undelete *
```

undeletes all deleted messages.

You can search for the presence of a word in subject lines with /. For example, to print the headers of all messages that contain the word "PASCAL," do:

```
from /pascal
```

Note that subject searching ignores upper/lower case differences.

5.2. List of commands

This section describes all the *Mail* commands available when receiving mail.

! Used to preface a command to be executed by the shell.

– The – command goes to the previous message and prints it. The – command may be given a decimal number *n* as an argument, in which case the *n*th previous message is gone to and printed.

Print
Like **print**, but also print out ignored header fields. See also **print** and **ignore**.

Reply
Note the capital R in the name. Frame a reply to a one or more messages. The reply (or replies if you are using this on multiple messages) will be sent ONLY to the person who sent you the message (respectively, the set of people who sent the messages you are replying to). You can add people using the **~t** and **~c** tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command. The **Reply** command is especially useful for replying to messages that were sent to enormous distribution groups when you really just want to send a message to the originator. Use it often.

Type
Identical to the **Print** command.

alias Define a name to stand for a set of other names. This is used when you want to send messages to a certain group of people and want to avoid retyping their names. For example

alias project john sue willie kathryn

creates an alias *project* which expands to the four people John, Sue, Willie, and Kathryn.

alternates
If you have accounts on several machines, you may find it convenient to use the /usr/lib/aliases on all the machines except one to direct your mail to a single account. The **alternates** command is used to inform *Mail* that each of these other addresses is really *you*. *Alternates* takes a list of user names and remembers that they are all actually you. When you **reply** to messages that were sent to one of these alternate names, *Mail* will not bother to send a copy of the message to this other address (which would simply be directed back to you by the alias mechanism). If *alternates* is given no argument, it lists the current set of alternate names. **Alternates** is usually used in the .mailrc file.

chdir
The **chdir** command allows you to change your current directory. **Chdir** takes a single argument, which is taken to be the pathname of the directory to change to. If no argument is given, **chdir** changes to your home directory.

copy The **copy** command does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.

delete
Deletes a list of messages. Deleted messages can be reclaimed with the **undelete** command.

dp These

commands delete the current message and print the next message. They are useful for quickly reading and disposing of mail.

edit To edit individual messages using the text editor, the **edit** command is provided. The **edit** command takes a list of messages as described under the **type** command and processes each by writing it into the file Message*x* where *x* is the message number being edited and executing the text editor on it. When you have edited the message to your satisfaction, write the message out and quit, upon which *Mail* will read the message back and remove the file. **Edit** may be abbreviated to **e**.

else Marks the end of the then-part of an **if** statement and the beginning of the part to take effect if the condition of the **if** statement is false.

endif Marks the end of an **if** statement.

exit Leave *Mail* without updating the system mailbox or the file your were reading. Thus, if you accidentally delete several messages, you can use **exit** to avoid scrambling your mailbox.

file The same as **folder**.

folders
List the names of the folders in your folder directory.

folder
The **folder** command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read the new file. Some special conventions are recognized for the name:

center; c c l a. Name Meaning _ # Previous file read % Your system mailbox %name *Name*'s system mailbox & Your ~/mbox file +folder A file in your folder directory

from The **from** command takes a list of messages and prints out the header lines for each one; hence

```
from joe
```

is the easy way to display all the message headers from "joe."

headers
When you start up *Mail* to read your mail, it lists the message headers that you have. These headers tell you who each message is from, when they were sent, how many lines and characters each message is, and the "Subject:" header field of each message, if present. In addition, *Mail* tags the message header of each message that has been the object of the **preserve** command with a "P." Messages that have been **saved** or **written** are flagged with a "*." Finally, **deleted** messages are not printed at all. If you wish to reprint the current list of message headers, you can do so with the **headers** command. The **headers** command (and thus the initial header listing) only lists the first so many message headers. The number of headers listed depends on the speed of your terminal. This can be overridden by specifying the number of headers you want with the *window* option. *Mail* maintains a notion of the current "window" into your messages for the purposes of printing headers. Use the **z** command to move forward and back a window. You can move *Mail's* notion of the current window directly to a particular message by using, for example,

```
headers 40
```

to move *Mail's* attention to the messages around message 40. The **headers** command can be abbreviated to **h**.

help Print a brief and usually out of date help message about the commands in *Mail*. The *man* page for *mail* is usually more up-to-date than either the help message or this manual.

hold Arrange to hold a list of messages in the system mailbox, instead of moving them to the file *mbox* in your home directory. If you set the binary option *hold*, this will happen by default.

if Commands in your ".mailrc" file can be executed conditionally depending on whether you are sending or receiving mail with the **if** command. For example, you can do:

```
if receive
        commands...
endif
```

An **else** form is also available:

```
if send
        commands...
else
        commands...
endif
```

Note that the only allowed conditions are **receive** and **send**.

ignore
Add the list of header fields named to the *ignore list*. Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header fields, such as *Via* which are not usually of interest. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

list List the vaild *Mail* commands.

mail Send mail to one or more people. If you have the *ask* option set, *Mail* will prompt you for a subject to your message. Then you can type in your message, using tilde escapes as described in section 4 to edit, print, or modify your message. To signal your satisfaction with the message and send it, type control-d at the beginning of a line, or a . alone on a line if you set the option *dot*. To abort the message, type two interrupt characters (RUBOUT by default) in a row or use the **~q** escape.

mbox
Indicate that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

next The **next** command goes to the next message and types it. If given a message list, **next** goes to the first such message and types it. Thus,

```
next root
```

goes to the next message sent by "root" and types it. The **next** command can be abbreviated to simply a newline, which means that one can go to and type a message by simply giving its message number or one of the magic characters "^" "." or "$". Thus,

```
.
```

prints the current message and

```
4
```

prints message 4, as described previously.

preserve
Same as **hold**. Cause a list of messages to be held in your system mailbox when you quit.

print Takes a message list and types out each message on the terminal.

quit Leave *Mail* and update the file, folder, or system mailbox your were reading. Messages that you have examined are marked as "read" and messages that existed when you started are marked as "old." If you were editing your system mailbox and if you have set the binary option *hold*, all messages which have not been deleted, saved, or mboxed will be retained in your system mailbox. If you were editing your system mailbox and you did *not* have *hold* set, all messages which have not been deleted, saved, or preserved will be moved to the file *mbox* in your home directory.

reply
Frame a reply to a single message. The reply will be sent to the person who sent you the message to which you are replying, plus all the people who received the original message, except you. You can add people using the **~t** and **~c** tilde escapes. The subject in your reply is formed by prefacing the subject in the original message with "Re:" unless it

already began thus. If the original message included a "reply-to" header field, the reply will go *only* to the recipient named by "reply-to." You type in your message using the same conventions available to you through the **mail** command.

save It is often useful to be able to save messages on related topics in a file. The **save** command gives you ability to do this. The **save** command takes as argument a list of message numbers, followed by the name of the file on which to save the messages. The messages are appended to the named file, thus allowing one to keep several messages in the file, stored in the order they were put there. The **save** command can be abbreviated to **s**. An example of the **save** command relative to our running example is:

s 1 2 tuitionmail

Saved messages are not automatically saved in *mbox* at quit time, nor are they selected by the **next** command described above, unless explicitly specified.

set Set an option or give an option a value. Used to customize *Mail*. Section 5.3 contains a list of the options. Options can be *binary*, in which case they are *on* or *off*, or *valued*. To set a binary option *option on*, do

set option

To give the valued option *option* the value *value*, do

set option=value

Several options can be specified in a single **set** command.

shell The **shell** command allows you to escape to the shell. **Shell** invokes an interactive shell and allows you to type commands to it. When you leave the shell, you will return to *Mail*. The shell used is a default assumed by *Mail*; you can override this default by setting the valued option "SHELL," eg:

set SHELL=/bin/csh

source
The **source** command reads *Mail* commands from a file. It is useful when you are trying to fix your ".mailrc" file and you need to re-read it.

top The **top** command takes a message list and prints the first five lines of each addressed message. It may be abbreviated to **to**. If you wish, you can change the number of lines that **top** prints out by setting the valued option "toplines." On a CRT terminal,

set toplines=10

might be preferred.

type Print a list of messages on your terminal. If you have set the option *crt* to a number and the total number of lines in the messages you are printing exceed that specified by *crt*, the messages will be printed by a terminal paging program such as *more*.

undelete
The **undelete** command causes a message that had been deleted previously to regain its initial status. Only messages that have been deleted may be undeleted. This command may be abbreviated to **u**.

unset
Reverse the action of setting a binary or valued option.

visual
It is often useful to be able to invoke one of two editors, based on the type of terminal one is using. To invoke a display oriented editor, you can use the **visual** command. The opera-

tion of the **visual** command is otherwise identical to that of the **edit** command.

Both the **edit** and **visual** commands assume some default text editors. These default editors can be overridden by the valued options "EDITOR" and "VISUAL" for the standard and screen editors. You might want to do:

set EDITOR=/usr/ucb/ex VISUAL=/usr/ucb/vi

write

The **save** command always writes the entire message, including the headers, into the file. If you want to write just the message itself, you can use the **write** command. The **write** command has the same syntax as the **save** command, and can be abbreviated to simply **w**. Thus, we could write the second message by doing:

w 2 file.c

As suggested by this example, the **write** command is useful for such tasks as sending and receiving source program text over the message system.

z *Mail* presents message headers in windowfuls as described under the **headers** command. You can move *Mail's* attention forward to the next window by giving the

z+

command. Analogously, you can move to the previous window with:

z−

5.3. Custom options

Throughout this manual, we have seen examples of binary and valued options. This section describes each of the options in alphabetical order, including some that you have not seen yet. To avoid confusion, please note that the options are either all lower case letters or all upper case letters. When I start a sentence such as: "Ask" causes *Mail* to prompt you for a subject header, I am only capitalizing "ask" as a courtesy to English.

EDITOR

The valued option "EDITOR" defines the pathname of the text editor to be used in the **edit** command and ~e. If not defined, a standard editor is used.

SHELL

The valued option "SHELL" gives the path name of your shell. This shell is used for the ! command and ~! escape. In addition, this shell expands file names with shell metacharacters like * and ? in them.

VISUAL

The valued option "VISUAL" defines the pathname of your screen editor for use in the **visual** command and ~v escape. A standard screen editor is used if you do not define one.

append

The "append" option is binary and causes messages saved in *mbox* to be appended to the end rather than prepended. Normally, *Mail*will *mbox* in the same order that the system puts messages in your system mailbox. By setting "append," you are requesting that *mbox* be appended to regardless. It is in any event quicker to append.

ask "Ask" is a binary option which causes *Mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

askcc

"Askcc" is a binary option which causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline shows your satisfaction with the current list.

autoprint
"Autoprint" is a binary option which causes the **delete** command to behave like **dp** — thus, after deleting a message, the next one will be typed automatically. This is useful to quickly scanning and deleting messages in your mailbox.

debug
The binary option "debug" causes debugging information to be displayed. Use of this option is the same as useing the

–d command line flag.

dot "Dot" is a binary option which, if set, causes *Mail* to interpret a period alone on a line as the terminator of a message you are sending.

escape
To allow you to change the escape character used when sending mail, you can set the valued option "escape." Only the first character of the "escape" option is used, and it must be doubled if it is to appear as the first character of a line of your message. If you change your escape character, then ~ loses all its special meaning, and need no longer be doubled at the beginning of a line.

folder
The name of the directory to use for storing folders of messages. If this name begins with a '/' *Mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

hold The binary option "hold" causes messages that have been read but not manually dealt with to be held in the system mailbox. This prevents such messages from being automatically swept into your mbox.

ignore
The binary option "ignore" causes RUBOUT characters from your terminal to be ignored and echoed as @'s while you are sending mail. RUBOUT characters retain their original meaning in *Mail* command mode. Setting the "ignore" option is equivalent to supplying the **–i** flag on the command line as described in section 6.

ignoreeof
An option related to "dot" is "ignoreeof" which makes *Mail* refuse to accept a control–d as the end of a message. "Ignoreeof" also applies to *Mail* command mode.

keep The "keep" option causes *Mail* to truncate your system mailbox instead of deleting it when it is empty. This is useful if you elect to protect your mailbox, which you would do with the shell command:

chmod 600 /usr/spool/mail/yourname

where *yourname* is your login name. If you do not do this, anyone can probably read your mail, although people usually don't.

keepsave
When you **save** a message, *Mail* usually discards it when you **quit**. To retain all saved messages, set the "keepsave" option.

metoo
When sending mail to an alias, *Mail* makes sure that if you are included in the alias, that mail will not be sent to you. This is useful if a single alias is being used by all members of the group. If however, you wish to receive a copy of all the messages you send to the alias, you can set the binary option "metoo."

noheader
The binary option "noheader" suppresses the printing of the version and headers when *Mail* is first invoked. Setting this option is the same as using **–N** on the command line.

nosave
Normally, when you abort a message with two RUBOUTs, *Mail* copies the partial letter to the file "dead.letter" in your home directory. Setting the binary option "nosave" prevents this.

quiet The binary option "quiet" suppresses the printing of the version when *Mail* is first invoked, as well as printing the for example "Message 4:" from the **type** command.

record
If you love to keep records, then the valued option "record" can be set to the name of a file to save your outgoing mail. Each new message you send is appended to the end of the file.

screen
When *Mail* initially prints the message headers, it determines the number to print by looking at the speed of your terminal. The faster your terminal, the more it prints. The valued option "screen" overrides this calculation and specifies how many message headers you want printed. This number is also used for scrolling with the **z** command.

sendmail
To alternate delivery system, set the "sendmail" option to the full pathname of the program to use. Note: this is not for everyone! Most people should use the default delivery system.

toplines
The valued option "toplines" defines the number of lines that the "top" command will print out instead of the default five lines.

verbose
The binary option "verbose" causes *Mail* to invoke sendmail with the **–v** flag, which causes it to go into versbose mode and announce expansion of aliases, etc. Setting the "verbose" option is equivalent to invoking *Mail* with the **–v** flag as described in section 6.

6. Command line options

This section describes command line options for *Mail* and what they are used for.

-N Suppress the initial printing of headers.

-d Turn on debugging information. Not of general interest.

-f fileShow the messages in *file* instead of your system mailbox. If *file* is omitted, *Mail* reads *mbox* in your home directory.

-i Ignore tty interrupt signals. Useful on noisy phone lines, which generate spurious RUBOUT or DELETE characters. It's usually more effective to change your interrupt character to control-c, for which see the *stty* shell command.

-n Inhibit reading of /usr/lib/Mail.rc. Not generally useful, since /usr/lib/Mail.rc is usually empty.

-s string
Used for sending mail. *String* is used as the subject of the message being composed. If *string* contains blanks, you must surround it with quote marks.

-u name
Read *names's* mail instead of your own. Unwitting others often neglect to protect their mailboxes, but discretion is advised. Essentially, **-u user** is a shorthand way of doing **-f /usr/spool/user**.

-v Use the **-v** flag when invoking sendmail. This feature may also be enabled by setting the the option "verbose".

The following command line flags are also recognized, but are intended for use by programs invoking *Mail* and not for people.

-T file
Arrange to print on *file* the contents of the *article-id* fields of all messages that were either read or deleted. **-T** is for the *readnews* program and should NOT be used for reading your mail.

-h number
Pass on hop count information. *Mail* will take the number, increment it, and pass it with **-h** to the mail delivery system. **-h** only has effect when sending mail and is used for network mail forwarding.

-r name
Used for network mail forwarding: interpret *name* as the sender of the message. The *name* and **-r** are simply sent along to the mail delivery system. Also, *Mail* will wait for the message to be sent and return the exit status. Also restricts formatting of message.

Note that **-h** and **-r**, which are for network mail forwarding, are not used in practice since mail forwarding is now handled separately. They may disappear soon.

7. Format of messages

This section describes the format of messages. Messages begin with a *from* line, which consists of the word "From" followed by a user name, followed by anything, followed by a date in the format returned by the *ctime* library routine described in section 3 of the Unix Programmer's Manual. A possible *ctime* format date is:

Tue Dec 1 10:58:23 1981

The *ctime* date may be optionally followed by a single space and a time zone indication, which should be three capital letters, such as PDT.

Following the *from* line are zero or more *header field* lines. Each header field line is of the form:

name: information

Name can be anything, but only certain header fields are recognized as having any meaning. The recognized header fields are: *article-id, bcc, cc, from, reply-to, sender, subject,* and *to.* Other header fields are also significant to other systems; see, for example, the current Arpanet message standard for much more on this topic. A header field can be continued onto following lines by making the first character on the following line a space or tab character.

If any headers are present, they must be followed by a blank line. The part that follows is called the *body* of the message, and must be ASCII text, not containing null characters. Each line in the message body must be terminated with an ASCII newline character and no line may be longer than 512 characters. If binary data must be passed through the mail system, it is suggested that this data be encoded in a system which encodes six bits into a printable character. For example, one could use the upper and lower case letters, the digits, and the characters comma and period to make up the 64 characters. Then, one can send a 16-bit binary number as three characters. These characters should be packed into lines, preferably lines about 70 characters long as long lines are transmitted more efficiently.

The message delivery system always adds a blank line to the end of each message. This blank line must not be deleted.

The UUCP message delivery system sometimes adds a blank line to the end of a message each time it is forwarded through a machine.

It should be noted that some network transport protocols enforce limits to the lengths of messages.

8. Glossary

This section contains the definitions of a few phrases peculiar to *Mail*.

alias An alternative name for a person or list of people.

flag An option, given on the command line of *Mail*, prefaced with a –. For example, **–f** is a flag.

header field
At the beginning of a message, a line which contains information that is part of the structure of the message. Popular header fields include *to*, *cc*, and *subject*.

mail
A collection of messages. Often used in the phrase, "Have you read your mail?"

mailbox
The place where your mail is stored, typically in the directory /usr/spool/mail.

message
A single letter from someone, initially stored in your *mailbox*.

message list
A string used in *Mail* command mode to describe a sequence of messages.

option
A piece of special purpose information used to tailor *Mail* to your taste. Options are specified with the **set** command.

9. Summary of commands, options, and escapes

This section gives a quick summary of the *Mail* commands, binary and valued options, and tilde escapes.

The following table describes the commands:

Command	*Description*
!	Single command escape to shell
-	Back up to previous message
Print	Type message with ignored fields
Reply	Reply to author of message only
Type	Type message with ignored fields
alias	Define an alias as a set of user names
alternates	List other names you are known by
chdir	Change working directory, home by default
copy	Copy a message to a file or folder
delete	Delete a list of messages
dt	Delete current message, type next message
endif	End of conditional statement; see **if**
edit	Edit a list of messages
else	Start of else part of conditional; see **if**
exit	Leave mail without changing anything
file	Interrogate/change current mail file
folder	Same as **file**
folders	List the folders in your folder directory
from	List headers of a list of messages
headers	List current window of messages
help	Print brief summary of *Mail* commands
hold	Same as **preserve**
if	Conditional execution of *Mail* commands
ignore	Set/examine list of ignored header fields
list	List valid *Mail* commands
local	List other names for the local host
mail	Send mail to specified names
mbox	Arrange to save a list of messages in *mbox*
next	Go to next message and type it
preserve	Arrange to leave list of messages in system mailbox
quit	Leave *Mail*; update system mailbox, *mbox* as appropriate
reply	Compose a reply to a message
save	Append messages, headers included, on a file
set	Set binary or valued options
shell	Invoke an interactive shell
top	Print first so many (5 by default) lines of list of messages
type	Print messages
undelete	Undelete list of messages
unset	Undo the operation of a **set**
visual	Invoke visual editor on a list of messages
write	Append messages to a file, don't include headers
z	Scroll to next/previous screenful of headers

The following table describes the options. Each option is shown as being either a binary or valued option.

Option	*Type*	*Description*
EDITOR	*valued*	Pathname of editor for ~e and **edit**
SHELL	*valued*	Pathname of shell for **shell**, ~! and **!**
VISUAL	*valued*	Pathname of screen editor for ~v, **visual**
append	*binary*	Always append messages to end of *mbox*
ask	*binary*	Prompt user for Subject: field when sending
askcc	*binary*	Prompt user for additional Cc's at end of message
autoprint	*binary*	Print next message after **delete**
crt	*valued*	Minimum number of lines before using *more*
debug	*binary*	Print out debugging information
dot	*binary*	Accept . alone on line to terminate message input
escape	*valued*	Escape character to be used instead of ~
folder	*valued*	Directory to store folders in
hold	*binary*	Hold messages in system mailbox by default
ignore	*binary*	Ignore RUBOUT while sending mail
ignoreeof	*binary*	Don't terminate letters/command input with ↑**D**
keep	*binary*	Don't unlink system mailbox when empty
keepsave	*binary*	Don't delete **saved** messages by default
metoo	*binary*	Include sending user in aliases
noheader	*binary*	Suppress initial printing of version and headers
nosave	*binary*	Don't save partial letter in *dead.letter*
quiet	*binary*	Suppress printing of *Mail* version and message numbers
record	*valued*	File to save all outgoing mail in
screen	*valued*	Size of window of message headers for **z**, etc.
sendmail	*valued*	Choose alternate mail delivery system
toplines	*valued*	Number of lines to print in **top**
verbose	*binary*	Invoke sendmail with the **–v** flag

The following table summarizes the tilde escapes available while sending mail.

Escape	*Arguments*	*Description*
~!	*command*	Execute shell command
~c	*name ...*	Add names to Cc: field
~d		Read *dead.letter* into message
~e		Invoke text editor on partial message
~f	*messages*	Read named messages
~h		Edit the header fields
~m	*messages*	Read named messages, right shift by tab
~p		Print message entered so far
~q		Abort entry of letter; like RUBOUT
~r	*filename*	Read file into message
~s	*string*	Set Subject: field to *string*
~t	*name ...*	Add names to To: field
~v		Invoke screen editor on message
~w	*filename*	Write message on file
~|	*command*	Pipe message through *command*
~~	*string*	Quote a ~ in front of *string*

The following table shows the command line flags that *Mail* accepts:

Flag	Description
–N	Suppress the initial printing of headers
–T *file*	Article-id's of read/deleted messages to *file*
–d	Turn on debugging
–f *file*	Show messages in *file* or *~/mbox*
–h *number*	Pass on hop count for mail forwarding
–i	Ignore tty interrupt signals
–n	Inhibit reading of /usr/lib/Mail.rc
–r *name*	Pass on *name* for mail forwarding
–s *string*	Use *string* as subject in outgoing mail
–u *name*	Read *name's* mail instead of your own
–v	Invoke sendmail with the **–v** flag

Notes: **–T**, **–d**, **–h**, and **–r** are not for human use.

Part 2
Text Editing

Awk — A Pattern Scanning and Processing Language (Second Edition)

Alfred V. Aho

Brian W. Kernighan

Peter J. Weinberger

ABSTRACT

Awk is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

```
length > 72
```

prints all input lines whose length exceeds 72 characters; the program

```
NF % 2 == 0
```

prints all lines with an even number of fields; and the program

```
{ $1 = log($1); print }
```

replaces the first field of each line by its logarithm.

Awk patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, **if-else**, **while**, **for** statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

1. Introduction

Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep* unix program manual will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

† UNIX is a trademark of AT&T Bell Laboratories.

```
$1 != prev    { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

1.1. Usage

The command

```
awk  program  [files]
```

executes the *awk* commands in the string **program** on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file **pfile**, and executed by the command

```
awk  -f pfile  [files]
```

1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern    { action }
pattern    { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

1.3. Records and Fields

Awk input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named **NR**.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as **$1, $2,** and so forth, where **$1** is the first field, and **$0** is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named **NF**.

The variables **FS** and **RS** refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument **–F***c* may also be used to set **FS** to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable **FILENAME** contains the name of the current input file.

1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command **print**. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables **NF** and **NR** can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, **$1**, on the file **foo1**, and the second field on file **foo2**. The **>>** notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file **foo**. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to **bwk**.

The variables **OFS** and **ORS** may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the **print** statement.

Awk also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in **format** and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8 digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of **printf** is identical to that used with C. C programm language prentice hall 1978

2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

2.1. BEGIN and END

The special pattern **BEGIN** matches the beginning of the input, before the first record is read. The pattern **END** matches the end of the input, after the last record has been processed. **BEGIN** and **END** thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN  { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END  { print NR }
```

If **BEGIN** is present, it must be the first pattern; **END** must be the last if used.

2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

Awk regular expressions include the regular expression forms found in the UNIX text editor *ed* unix program manual and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in *lex*. Character classes may be abbreviated: **[a–zA–Z0–9]** is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly **[jJ]ohn**, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an **s**, **t**, **u**, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1>= "s"&& $1<"t"&& $1!="smith"
```

selects lines where the first field begins with "s", but is not "smith". && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2    { ... }
```

In this case, the action is performed for each line between an occurrence of **pat1** and the next occurrence of **pat2** (inclusive). For example,

```
/start/, /stop/
```

prints all lines between **start** and **stop**, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

3.1. Built-in Functions

Awk provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

Awk also provides the arithmetic functions **sqrt**, **log**, **exp**, and **int**, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **substr(s, m, n)** produces the substring of **s** that begins at position **m** (origin 1) and is at most **n** characters long. If **n** is omitted, the substring goes to the end of **s**. The function **index(s1, s2)** returns the position where the string **s2** occurs in **s1**, or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions **e1**, **e2**, etc., in the **printf** format specified by **f**. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets x to the string produced by formatting the values of $1 and $2.

3.2. Variables, Expressions, and Assignments

Awk variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to x. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by

```
    { s1 += $1; s2 += $2 }
END     { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). The C increment ++ and decrement — operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=. These operators may all be used in expressions.

3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
    $3 = "too big"
  print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string **s** into **array**[1], ..., **array**[n]. The number of elements found is returned. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
    { x[NR] = $0 }
END     { ... program ... }
```

The first action merely records each input line in the array **x**.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like **apple**, **orange**, etc. Then the program

```
/apple/  { x["apple"]++ }
/orange/ { x["orange"]++ }
END
{ print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

3.6. Flow-of-Control Statements

Awk provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in C. We showed the **if** statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the **if** is done. The **else** part is optional.

The **while** statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The **for** statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
    print $i
```

does the same job as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

for (i in array)
statement

does *statement* with **i** set in turn to each element of **array**. The elements are accessed in an apparently random order. Chaos will ensue if **i** is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while** or **for** can include relational operators like <, <=, >, >=, == ("is equal to"), and != ("not equal to"); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin.

The statement **next** causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y     # this is a comment
```

4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed* unix programm manual provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

Lex lesk lexical analyzer cstr provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

Awk is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

Awk also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*; yacc johnson cstr the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

Awk was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".

3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number : ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls -l*; each line has the form

-rw-rw-rw- 1 ava 123 Oct 15 17:05 xxx

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*. $LIST$

Program	Task 1	2	3	4	5	6	7	8
wc	8.6							
grep	11.7	13.1						
egrep	6.2	11.5	11.6					
fgrep	7.7	13.8	16.1					
sed	10.2	11.6	15.8	29.0	30.5	16.1		
lex	65.1	150.1	144.2	67.7	70.3	104.0	81.7	92.8
awk	15.0	25.6	29.9	33.3	38.9	46.4	71.4	31.1

Table I. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. `END {print NR}`

2. `/doug/`

3. `/ken|doug|dmr/`

4. `{print $3}`

5. `{print $3, $2}`

6. `/ken/ {print >"jken"}`
   `/doug/ {print >"jdoug"}`
   `/dmr/ {print >"jdmr"}`

7. `{print NR ": " $0}`

8. `{sum = sum + $4}`
   `END {print sum}`

SED:

1. `$=`

2. `/doug/p`

3. `/doug/p`
   `/doug/d`
   `/ken/p`
   `/ken/d`
   `/dmr/p`
   `/dmr/d`

4. `/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p`

5. `/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p`

6. `/ken/w jken`
   `/doug/w jdoug`
   `/dmr/w jdmr`

LEX:

1.
```
%{
int i;
%}
%%
\n    i++;
.     ;
%%
yywrap() {
      printf("%d\n", i);
}
```

2.
```
%%
^.*doug.*$      printf("%s\n", yytext);
.     ;
\n    ;
```

Ex Reference Manual
Version 3.7

William Joy

Mark Horton

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, Ca. 94720

ABSTRACT

Ex a line oriented text editor, which supports both command and display oriented editing. This reference manual describes the command oriented part of *ex;* the display editing features of *ex* are described in *An Introduction to Display Editing with Vi.* Other documents about the editor include the introduction *Edit: A tutorial*, the *Ex/edit Command Summary*, and a *Vi Quick Reference* card.

1. Starting ex

Each instance of the editor has a set of options, which can be set to tailor it to your liking. The command *edit* invokes a version of *ex* designed for more casual or beginning users by changing the default settings of some of these options. To simplify the description which follows we assume the default settings of the options.

When invoked, *ex* determines the terminal type from the TERM variable in the environment. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, then that description is used. Also if the TERMCAP variable contains a pathname (beginning with a /) then the editor will seek the description of the terminal in that file (rather than the default /etc/termcap). If there is a variable EXINIT in the environment, then the editor will execute the commands in that variable, otherwise if there is a file *.exrc* in your HOME directory *ex* reads commands from that file, simulating a *source* command. Option setting commands placed in EXINIT or *.exrc* will be executed before each editor session.

A command to enter *ex* has the following prototype:†

ex [**–**] [**–v**] [**–t** *tag*] [**–r**] [**–l**] [**–w***n*] [**–x**] [**–R**] [**+***command*] name ...

The most common case edits a single file with no options, i.e.:

ex name

The **–** command line option option suppresses all interactive-user feedback and is useful in processing editor scripts in command files. The **–v** option is equivalent to using *vi* rather than *ex.* The **–t** option is equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition. The **–r** option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, typing a list of saved files. The **–l** option sets up for editing LISP, setting the *showmatch* and *lisp* options.

The financial support of an IBM Graduate Fellowship and the National Science Foundation under grants MCS74-07644-A03 and MCS78-07291 is gratefully acknowledged.

† Brackets '[' ']' surround optional parameters here.

The **–w** option sets the default window size to *n,* and is useful on dialups to start in small windows. The **–x** option causes *ex* to prompt for a *key*, which is used to encrypt and decrypt the contents of the file, which should already be encrypted using the same key, see *crypt*(1). The **–R** option sets the *readonly* option at the start. *Name* arguments indicate files to be edited. An argument of the form +*command* indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form "/pat" or line numbers, e.g. "+100" starting at line 100.

2. File manipulation

2.1. Current file

Ex is normally editing the contents of a single file, whose name is recorded in the *current* file name. *Ex* performs all editing actions in a buffer (actually a temporary file) into which the text of the file is initially read. Changes made to the buffer have no effect on the file being edited unless and until the buffer contents are written out to the file with a *write* command. After the buffer contents are written, the previous contents of the written file are no longer accessible. When a file is edited, its name becomes the current file name, and its contents are read into the buffer.

The current file is almost always considered to be *edited.* This means that the contents of the buffer are logically connected with the current file name, so that writing the current buffer contents onto that file, even if it exists, is a reasonable action. If the current file is not *edited* then *ex* will not normally write on it if it already exists.*

2.2. Alternate file

Each time a new value is given to the current file name, the previous current file name is saved as the *alternate* file name. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate file name.

2.3. Filename expansion

Filenames within the editor may be specified using the normal shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* file name and the character '#' by the *alternate* file name.†

2.4. Multiple files and named buffers

If more than one file is given on the command line, then the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list.* The current argument list may be displayed with the *args* command. The next file in the argument list may be edited with the *next* command. The argument list may also be respecified by specifying a list of names to the *next* command. These names are expanded, the resulting list of names becomes the new argument list, and *ex* edits the first file on the list.

For saving blocks of text while editing, and especially when editing more than one file, *ex* has a group of named buffers. These are similar to the normal buffer, except that only a limited number of operations are available on them. The buffers have names *a* through *z.*‡

---

* The *file* command will say "[Not edited]" if the current file is not considered edited.

† This makes it easy to deal alternately with two files and eliminates the need for retyping the name supplied on an *edit* command after a *No write since last change* diagnostic is received.

‡ It is also possible to refer to *A* through *Z;* the upper case buffers are the same as the lower but commands append to named buffers rather than replacing if upper case names are used.

2.5. Read only

It is possible to use *ex* in *read only* mode to look at files that you have no intention of modifying. This mode protects you from accidently overwriting the file. Read only mode is on when the *readonly* option is set. It can be turned on with the –R command line option, by the *view* command line invocation, or by setting the *readonly* option. It can be cleared by setting *noreadonly*. It is possible to write, even while in read only mode, by indicating that you really know what you are doing. You can write to a different file, or can use the ! form of write, even while in read only mode.

3. Exceptional Conditions

3.1. Errors and interrupts

When errors occur *ex* (optionally) rings the terminal bell and, in any case, prints an error diagnostic. If the primary input is from a file, editor processing will terminate. If an interrupt signal is received, *ex* prints "Interrupt" and returns to its command level. If the primary input is a file, then *ex* will exit when this occurs.

3.2. Recovering from hangups and crashes

If a hangup signal is received and the buffer has been modified since it was last written out, or if the system crashes, either the editor (in the first case) or the system (after it reboots in the second) will attempt to preserve the buffer. The next time you log in you should be able to recover the work you were doing, losing at most a few lines of changes from the last point before the hangup or editor crash. To recover a file you can use the –r option. If you were editing the file *resume*, then you should change to the directory where you were when the crash occurred, giving the command

ex –r *resume*

After checking that the retrieved file is indeed ok, you can *write* it over the previous contents of that file.

You will normally get mail from the system telling you when a file has been saved after a crash. The command

ex –r

will print a list of the files which have been saved for you. (In the case of a hangup, the file will not appear in the list, although it can be recovered.)

4. Editing modes

Ex has five distinct modes. The primary mode is *command* mode. Commands are entered in command mode when a ':' prompt is present, and are executed each time a complete line is sent. In *text input* mode *ex* gathers input lines and places them in the file. The *append, insert,* and *change* commands use text input mode. No prompt is printed when you are in text input mode. This mode is left by typing a '.' alone at the beginning of a line, and *command* mode resumes.

The last three modes are *open* and *visual* modes, entered by the commands of the same name, and, within open and visual modes *text insertion* mode. *Open* and *visual* modes allow local editing operations to be performed on the text in the file. The *open* command displays one line at a time on any terminal while *visual* works on CRT terminals with random positioning cursors, using the screen as a (single) window for file editing changes. These modes are described (only) in *An Introduction to Display Editing with Vi.*

5. Command structure

Most command names are English words, and initial prefixes of the words are acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands.*

5.1. Command parameters

Most commands accept prefix addresses specifying the lines in the file upon which they are to have effect. The forms of these addresses will be discussed below. A number of commands also may take a trailing *count* specifying the number of lines to be involved in the command.† Thus the command "10p" will print the tenth line in the buffer while "delete 5" will delete five lines from the buffer, starting with the current line.

Some commands take other information or parameters, this information always being given after the command name.‡

5.2. Command variants

A number of commands have two distinct variants. The variant form of the command is invoked by placing an '!' immediately after the command name. Some of the default variants may be controlled by options; in this case, the '!' serves to toggle the default.

5.3. Flags after commands

The characters '#', 'p' and 'l' may be placed after many commands.** In this case, the command abbreviated by these characters is executed after the command completes. Since *ex* normally prints the new current line after each change, 'p' is rarely necessary. Any number of '+' or '-' characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

5.4. Comments

It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. The comment character is the double quote: ". Any command line beginning with " is ignored. Comments beginning with " may also be placed at the ends of commands, except in cases where they could be confused as part of text (shell escapes and the substitute and map commands).

5.5. Multiple commands per line

More than one command may be placed on a line by separating each pair of commands by a '|' character. However the *global* commands, comments, and the shell escape '!' must be the last command on a line, as they are not terminated by a '|'.

5.6. Reporting large changes

Most commands which change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the *report* option. This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with an *undo*. After commands with more global effect such as *global* or *visual*, you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

---

* As an example, the command *substitute* can be abbreviated 's' while the shortest available abbreviation for the *set* command is 'se'.

† Counts are rounded down if necessary.

‡ Examples would be option names in a *set* command i.e. "set number", a file name in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, i.e. "1,5 copy 25".

** A 'p' or 'l' must be preceded by a blank or tab except in the single special case 'dp'.

6. Command addressing

6.1. Addressing primitives

.	The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus '.' is rarely used alone as an address.
n	The *n*th line in the editor's buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
+n -n	An offset relative to the current buffer line.†
/pat/ ?pat?	Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below). The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing *pat*, then the trailing / or ? may be omitted. If *pat* is omitted or explicitly empty, then the last regular expression specified is located.‡
'' 'x	Before each non-relative motion of the current line '.', the previous current line is marked with a tag, subsequently referred to as ''''. This makes it easy to refer or return to this previous context. Marks may also be established by the *mark* command, using single lower case letters *x* and the marked lines referred to as *''x*'.

6.2. Combining addressing primitives

Addresses to commands consist of a series of addressing primitives, separated by ',' or ';'. Such address lists are evaluated left-to-right. When addresses are separated by ';' the current line '.' is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer.†

7. Command descriptions

The following form is a prototype for all *ex* commands:

address **command** *! parameters count flags*

All parts are optional; the degenerate case is the empty command which prints the next line in the file. For sanity with use from within *visual* mode, *ex* ignores a ":" preceding any command.

In the following command descriptions, the default addresses are shown in parentheses, which are *not,* however, part of the command.

abbreviate *word rhs* abbr: **ab**

Add the named abbreviation to the current list. When in input mode in visual, if *word* is typed as a complete word, it will be changed to *rhs*.

† The forms '.+3' '+3' and '+++' are all equivalent; if the current line is line 100 they all address line 103.

‡ The forms \/ and \? scan using the last regular expression used in a scan; after a substitute // and ?? would scan using the substitute's regular expression.

† Null address specifications are permitted in a list of addresses, the default in this case is the current line '.'; thus ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command which expects none.

(.) **append** abbr: **a**
text
.

Reads the input text and places it after the specified line. After the command, '.' addresses the last line input or the specified line if no lines were input. If address '0' is given, text is placed at the beginning of the buffer.

a!
text
.

The variant flag to *append* toggles the setting for the *autoindent* option during the input of *text.*

args

The members of the argument list are printed, with the current argument delimited by '[' and ']'.

(. , .) **change** *count* abbr: **c**
text
.

Replaces the specified lines with the input *text.* The current line becomes the last line input; if no lines were input it is left as for a *delete.*

c!
text
.

The variant toggles *autoindent* during the *change.*

(. , .) **copy** *addr flags* abbr: **co**

A *copy* of the specified lines is placed after *addr,* which may be '0'. The current line '.' addresses the last line of the copy. The command *t* is a synonym for *copy.*

(. , .) **delete** *buffer count flags* abbr: **d**

Removes the specified lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line. If a named *buffer* is specified by giving a letter, then the specified lines are saved in that buffer, or appended to it if an upper case letter is used.

edit *file* abbr: **e**
ex *file*

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last *write* command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After insuring that this file is sensible† the editor reads the file into its buffer.

If the read of the file completes without error, the number of lines and characters read is typed. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered *edited.* If the last line of the input file is missing the trailing newline

† I.e., that it is not a binary file such as a directory, a block or character special file other than */dev/tty,* a terminal, or a binary or executable file (as indicated by the first word).

character, it will be supplied and a complaint will be issued. This command leaves the current line '.' at the last line read.‡

e! *file*

The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e +*n* *file*

Causes the editor to begin at line *n* rather than at the last line; *n* may also be an editor command containing no spaces, e.g.: "+/pat".

file abbr: **f**

Prints the current file name, whether it has been '[Modified]' since the last *write* command, whether it is *read only*, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line.*

file *file*

The current file name is changed to *file* which is considered '[Not edited]'.

(1 , $) global */pat/ cmds* abbr: **g**

First marks each line among those specified which matches the given regular expression. Then the given command list is executed with '.' initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a '\'. If *cmds* (and possibly the trailing / delimiter) is omitted, each line matching *pat* is printed. *Append, insert,* and *change* commands and associated input are permitted; the '.' terminating input may be omitted if it would be on the last line of the command list. *Open* and *visual* commands are permitted in the command list and take input from the terminal.

The *global* command itself may not appear in *cmds*. The *undo* command is also not permitted there, as *undo* instead can be used to reverse the entire *global* command. The options *autoprint* and *autoindent* are inhibited during a *global,* (and possibly the trailing / delimiter) and the value of the *report* option is temporarily infinite, in deference to a *report* for the entire global. Finally, the context mark ''' is set to the value of '.' before the global command begins and is not changed during a global command, except perhaps by an *open* or *visual* within the *global.*

g! */pat/ cmds* abbr: **v**

The variant form of *global* runs *cmds* at each line not matching *pat.*

(.) insert abbr: **i**
text

Places the given text before the specified line. The current line is left at the last line input; if there were none input it is left at the line before the addressed line. This command differs from *append* only in the placement of text.

‡ If executed from within *open* or *visual,* the current line is initially the first line of the file.

* In the rare case that the current file is '[Not edited]' this is noted also; in this case you have to use the form **w!** to write to the file, since the editor is not sure that a **write** will not destroy a file unrelated to the current contents of the buffer.

i!
text
.

The variant toggles *autoindent* during the *insert*.

(. , .+1) **join** *count flags* abbr: **j**

Places the text from a specified range of lines together on one line. White space is adjusted at each junction to provide at least one blank character, two if there was a '.' at the end of the line, or none if the first following character is a ')'. If there is already white space at the end of the line, then the white space at the start of the next line will be discarded.

j!

The variant causes a simpler *join* with no white space processing; the characters in the lines are simply concatenated.

(.) **k** *x*

The *k* command is a synonym for *mark*. It does not require a blank or tab before the following letter.

(. , .) **list** *count flags*

Prints the specified lines in a more unambiguous way: tabs are printed as '^I' and the end of each line is marked with a trailing '$'. The current line is left at the last line printed.

map *lhs rhs*

The *map* command is used to define macros for use in *visual* mode. *Lhs* should be a single character, or the sequence "#n", for n a digit, referring to function key *n*. When this character or function key is typed in *visual* mode, it will be as though the corresponding *rhs* had been typed. On terminals without function keys, you can type "#n". See section 6.9 of the "Introduction to Display Editing with Vi" for more details.

(.) **mark** *x*

Gives the specified line mark *x*, a single lower case letter. The *x* must be preceded by a blank or a tab. The addressing form ''x' then addresses this line. The current line is not affected by this command.

(. , .) **move** *addr* abbr: **m**

The *move* command repositions the specified lines to be after *addr*. The first of the moved lines becomes the current line.

next abbr: **n**

The next file from the command line argument list is edited.

n!

The variant suppresses warnings about the modifications to the buffer not having been written out, discarding (irretrievably) any changes which may have been made.

n *filelist*
n +*command filelist*

The specified *filelist* is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If *command* is given (it must contain no spaces), then it is executed after editing the first such file.

(. , .) **number** *count flags* abbr: **#** or **nu**

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed.

(.) **open** *flags* abbr: **o**
(.) **open** */pat/ flags*

Enters intraline editing *open* mode at each addressed line. If *pat* is given, then the cursor will be placed initially at the beginning of the string matched by the pattern. To exit this mode use Q. See *An Introduction to Display Editing with Vi* for more details.

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a *write* command has resulted in an error and you don't know how to save your work. After a *preserve* you should seek help.

(. , .) **print** *count* abbr: **p** or **P**

Prints the specified lines with non-printing characters printed as control characters '^*x*'; delete (octal 177) is represented as '^?'. The current line is left at the last line printed.

(.) **put** *buffer* abbr: **pu**

Puts back previously *deleted* or *yanked* lines. Normally used with *delete* to effect movement of lines, or with *yank* to effect duplication of lines. If no *buffer* is specified, then the last *deleted* or *yanked* text is restored.* By using a named buffer, text may be restored that was saved there at any previous time.

quit abbr: **q**

Causes *ex* to terminate. No automatic write of the editor buffer to a file is performed. However, *ex* issues a warning message if the file has changed since the last *write* command was issued, and does not *quit.*† Normally, you will wish to save your changes, and you should give a *write* command; if you wish to discard them, use the **q!** command variant.

q!

Quits from the editor, discarding changes to the buffer without complaint.

(.) **read** *file* abbr: **r**

Places a copy of the text of the given file in the editing buffer after the specified line. If no *file* is given the current file name is used. The current file name is not changed unless there is none in which case *file* becomes the current name. The sensibility restrictions for the *edit* command apply here also. If the file buffer is empty and there is no current name then *ex* treats this as an *edit* command.

Address '0' is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the *edit* command when the *read* successfully terminates. After a *read* the current line is the last line read.‡

---

* But no modifying commands may intervene between the *delete* or *yank* and the *put,* nor may lines be moved between files without using a named buffer.

† *Ex* will also issue a diagnostic if there are more files in the argument list.

‡ Within *open* and *visual* the current line is set to the first line read rather than the last.

(.) **read** !*command*

Reads the output of the command *command* into the buffer after the specified line. This is not a variant form of the command, rather a read specifying a *command* rather than a *filename;* a blank or tab before the ! is mandatory.

recover *file*

Recovers *file* from the system save area. Used after a accidental hangup of the phone** or a system crash** or *preserve* command. Except when you use *preserve* you will be notified by mail when a file is saved.

rewind abbr: **rew**

The argument list is rewound, and the first file in the list is edited.

rew!

Rewinds the argument list discarding any changes made to the current buffer.

set *parameter*

With no arguments, prints those options whose values have been changed from their defaults; with parameter *all* it prints all of the option values.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is unnecessary unless the option is Boolean valued. Boolean options are given values either by the form 'set *option*' to turn them on or 'set no*option*' to turn them off; string and numeric options are assigned via the form 'set *option*=value'.

More than one parameter may be given to *set* ; they are interpreted left-to-right.

shell abbr: **sh**

A new shell is created. When it terminates, editing resumes.

source *file* abbr: **so**

Reads and executes commands from the specified file. *Source* commands may be nested.

(. , .) **substitute** */pat/repl/ options count flags* abbr: **s**

On each specified line, the first instance of pattern *pat* is replaced by replacement pattern *repl.* If the *global* indicator option character 'g' appears, then all instances are substituted; if the *confirm* indication character 'c' appears, then before each substitution the line to be substituted is typed with the string to be substituted marked with '↑' characters. By typing an 'y' one can cause the substitution to be performed, any other input causes no change to take place. After a *substitute* the current line is the last line substituted.

Lines may be split by substituting new-line characters into them. The newline in *repl* must be escaped by preceding it with a '\'. Other metacharacters available in *pat* and *repl* are described below.

stop

Suspends the editor, returning control to the top level shell. If *autowrite* is set and there are unsaved changes, a write is done first unless the form **stop!** is used. This commands is only available where supported by the teletype driver and operating system.

** The system saves a copy of the file you were editing only if you have made changes to the file.

(. , .) **substitute** *options count flags* abbr: **s**

If *pat* and *repl* are omitted, then the last substitution is repeated. This is a synonym for the & command.

(. , .) **t** *addr flags*

The *t* command is a synonym for *copy*.

ta *tag*

The focus of editing switches to the location of *tag*, switching to a different line in the current file where it is defined, or if necessary to another file.‡

The tags file is normally created by a program such as *ctags*, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag; this field is usually a contextual scan using '*/pat/*' to be immune to minor changes in the file. Such scans are always performed as if *nomagic* was set.

The tag names in the tags file must be sorted alphabetically.

unabbreviate *word* abbr: **una**

Delete *word* from the list of abbreviations.

undo abbr: **u**

Reverses the changes made in the buffer by the last buffer editing command. Note that *global* commands are considered a single command for the purpose of *undo* (as are *open* and *visual.)* Also, the commands *write* and *edit* which interact with the file system cannot be undone. *Undo* is its own inverse.

Undo always marks the previous value of the current line '.' as ''''. After an *undo* the current line is the first line restored or the line before the first line deleted if no lines were restored. For commands with more global effect such as *global* and *visual* the current line regains it's pre-command value after an *undo*.

unmap *lhs*

The macro expansion associated by *map* for *lhs* is removed.

(1 , $) **v** */pat/ cmds*

A synonym for the *global* command variant **g!**, running the specified *cmds* on each line which does not match *pat*.

version abbr: **ve**

Prints the current version number of the editor as well as the date the editor was last changed.

(.) **visual** *type count flags* abbr: **vi**

Enters visual mode at the specified line. *Type* is optional and may be '–' , '↑' or '.' as in the *z* command to specify the placement of the specified line on the screen. By default, if *type* is omitted, the specified line is placed as the first on the screen. A *count* specifies an initial window size; the default is the value of the option *window*. See the document *An Introduction to Display Editing with Vi* for more details. To exit this mode, type Q.

‡ If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no *tag* will reuse the previous tag.

visual file
visual +*n* file

From visual mode, this command is the same as edit.

(1 , $) **write** *file* abbr: **w**

Writes changes made back to *file*, printing the number of lines and characters written. Normally *file* is omitted and the text goes back where it came from. If a *file* is specified, then text will be written to that file.* If the file does not exist it is created. The current file name is changed only if there is no current file name; the current line is never changed.

If an error occurs while writing the current and *edited* file, the editor considers that there has been "No write since last change" even if the buffer had not previously been modified.

(1 , $) **write>>** *file* abbr: **w>>**

Writes the buffer contents at the end of an existing file.

w! *name*

Overrides the checking of the normal *write* command, and will write to any file which the system permits.

(1 , $) **w** !*command*

Writes the specified lines into *command*. Note the difference between **w!** which overrides checks and **w !** which writes to a command.

wq *name*

Like a *write* and then a *quit* command.

wq! *name*

The variant overrides checking on the sensibility of the *write* command, as **w!** does.

xit *name*

If any changes have been made and not written, writes the buffer out. Then, in any case, quits.

(. , .)**yank** *buffer count* abbr: **ya**

Places the specified lines in the named *buffer*, for later retrieval via *put*. If no buffer name is specified, the lines go to a more volatile place; see the *put* command description.

(.+**1**) **z** *count*

Print the next *count* lines, default *window*.

(.) **z** *type count*

Prints a window of text with the specified line at the top. If *type* is '–' the line is placed at the bottom; a '.' causes the line to be placed in the center.* A count gives the number of lines to be displayed rather than double the number specified by the *scroll* option. On a CRT the screen is cleared before display begins unless a count which is less than the screen size is given. The current line is left at the last line printed.

---

* The editor writes to a file only if it is the current file and is *edited*, if the file does not exist, or if the file is actually a teletype, */dev/tty*, */dev/null*. Otherwise, you must give the variant form **w!** to force the write.

* Forms 'z=' and 'z↑' also exist; 'z=' places the current line in the center, surrounds it with lines of '–' characters and leaves the current line at this line. The form 'z↑' prints the window before 'z–' would. The characters '+', '↑' and '–' may be repeated for cumulative effect. On some v2 editors, no *type* may be given.

! *command*

The remainder of the line after the '!' character is sent to a shell to be executed. Within the text of *command* the characters '%' and '#' are expanded as in filenames and the character '!' is replaced with the text of the previous command. Thus, in particular, '!!' repeats the last such shell escape. If any such expansion is performed, the expanded line will be echoed. The current line is unchanged by this command.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic will be printed before the command is executed as a warning. A single '!' is printed when the command completes.

(*addr* , *addr*) ! *command*

Takes the specified address range and supplies it as standard input to *command;* the resulting output then replaces the input lines.

($) =

Prints the line number of the addressed line. The current line is unchanged.

(. , .) > *count flags*
(. , .) < *count flags*

Perform intelligent shifting on the specified lines; < shifts left and > shift right. The quantity of shift is determined by the *shiftwidth* option and the repetition of the specification character. Only white space (blanks and tabs) is shifted; no non-white characters are discarded in a left-shift. The current line becomes the last line which changed due to the shifting.

^D

An end-of-file from a terminal input scrolls through the file. The *scroll* option specifies the size of the scroll, normally a half screen of text.

(.+1 , .+1)
(.+1 , .+1) |

An address alone causes the addressed lines to be printed. A blank line prints the next line in the file.

(. , .) & *options count flags*

Repeats the previous *substitute* command.

(. , .) ~ *options count flags*

Replaces the previous regular expression with the previous replacement pattern from a substitution.

8. Regular expressions and substitute replacement patterns

8.1. Regular expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. *Ex* remembers two previous regular expressions: the previous regular expression used in a *substitute* command and the previous regular expression used elsewhere (referred to as the previous *scanning* regular expression.) The previous regular expression can always be referred to by a null *re*, e.g. '//' or '??'.

8.2. Magic and nomagic

The regular expressions allowed by *ex* are constructed in one of two ways depending on the setting of the *magic* option. The *ex* and *vi* default setting of *magic* gives quick access to a powerful set of regular expression metacharacters. The disadvantage of *magic* is that the user must remember that these metacharacters are *magic* and precede them with the character '\' to use them as "ordinary" characters. With *nomagic,* the default for *edit,* regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the (now) ordinary character with a '\'. Note that '\' is thus always a metacharacter.

The remainder of the discussion of regular expressions assumes that that the setting of this option is *magic.*†

8.3. Basic regular expression summary

The following basic constructs are used to construct *magic* mode regular expressions.

char — An ordinary character matches itself. The characters '↑' at the beginning of a line, '$' at the end of line, '*' as any character other than the first, '.', '\', '[', and '~' are not ordinary characters and must be escaped (preceded) by '\' to be treated as such.

↑ — At the beginning of a pattern forces the match to succeed only at the beginning of a line.

$ — At the end of a regular expression forces the match to succeed only at the end of the line.

. — Matches any single character except the new-line character.

\< — Forces the match to occur only at the beginning of a "variable" or "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> — Similar to '\<', but matching the end of a "variable" or "word", i.e. either the end of the line or before character which is neither a letter, nor a digit, nor the underline character.

[*string*] — Matches any (single) character in the class defined by *string.* Most characters in *string* define themselves. A pair of characters separated by '–' in *string* defines the set of characters collating between the specified lower and upper bounds, thus '[a–z]' as a regular expression matches any (single) lower-case letter. If the first character of *string* is an '↑' then the construct matches those characters which it otherwise would not; thus '[↑a–z]' matches anything but a lower-case letter (and of course a newline). To place any of the characters '↑', '[', or '–' in *string* you must escape them with a preceding '\'.

8.4. Combining regular expression primitives

The concatenation of two regular expressions matches the leftmost and then longest string which can be divided with the first piece matching the first regular expression and the second piece matching the second. Any of the (single character matching) regular expressions mentioned above may be followed by the character '*' to form a regular expression which matches any number of adjacent occurrences (including 0) of characters matched by the regular expression it follows.

The character '~' may be used in a regular expression, and matches the text which defined the replacement part of the last *substitute* command. A regular expression may be enclosed

† To discern what is true with *nomagic* it suffices to remember that the only special characters in this case will be '↑' at the beginning of a regular expression, '$' at the end of a regular expression, and '\'. With *nomagic* the characters '~' and '&' also lose their special meanings related to the replacement pattern of a substitute.

between the sequences '\(' and '\)' with side effects in the *substitute* replacement patterns.

8.5. Substitute replacement patterns

The basic metacharacters for the replacement pattern are '&' and '~'; these are given as '\&' and '\~' when *nomagic* is set. Each instance of '&' is replaced by the characters which the regular expression matched. The metacharacter '~' stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by the escaping character '\'. The sequence '\n' is replaced by the text matched by the *n*-th regular subexpression enclosed between '\(' and '\)'.† The sequences '\u' and '\l' cause the immediately following character in the replacement to be converted to upper- or lower-case respectively if this character is a letter. The sequences '\U' and '\L' turn such conversion on, either until '\E' or '\e' is encountered, or until the end of the replacement pattern.

9. Option descriptions

autoindent, ai default: noai

Can be used to ease the preparation of structured program text. At the beginning of each *append*, *change* or *insert* command or when a new line is *opened* or created by an *append*, *change*, *insert*, or *substitute* operation within *open* or *visual* mode, *ex* looks at the line being appended after, the first line changed or the line inserted before and calculates the amount of white space at the start of the line. It then aligns the cursor at the level of indentation so determined.

If the user then types lines of text in, they will continue to be justified at the displayed indenting level. If more white space is typed at the beginning of a line, the following line will start aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop one can hit **^D**. The tab stops going backwards are defined at multiples of the *shiftwidth* option. You *cannot* backspace over the indent, except by sending an end-of-file with a **^D**.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the *autoindent* is discarded.) Also specially processed in this mode are lines beginning with an '↑' and immediately followed by a **^D**. This causes the input to be repositioned at the beginning of the line, but retaining the previous indent for the next line. Similarly, a '0' followed by a **^D** repositions at the beginning but without retaining the previous indent.

Autoindent doesn't happen in *global* commands or when the input is not a terminal.

autoprint, ap default: ap

Causes the current line to be printed after each *delete*, *copy*, *join*, *move*, *substitute*, *t*, *undo* or shift command. This has the same effect as supplying a trailing 'p' to each such command. *Autoprint* is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give a *next*, *rewind*, *stop*, *tag*, or *!* command, or a **^↑** (switch files) or **^]** (tag goto) command in *visual*. Note, that the *edit* and *ex* commands do **not** autowrite. In each case, there is an equivalent way of switching when autowrite is set to avoid the *autowrite* (*edit* for *next*, *rewind!* for .I rewind , *stop!* for *stop*, *tag!* for *tag*, *shell* for *!*, and **:e #** and a **:ta!**

† When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

command from within *visual).*

beautify, bf default: nobeautify

Causes all control characters except tab, newline and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. *Beautify* does not apply to command input.

directory, dir default: dir=/tmp

Specifies the directory in which *ex* places its buffer file. If this directory in not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible default: noedcompatible

Causes the presence of absence of **g** and **c** suffixes on substitute commands to be remembered, and to be toggled by repeating the suffices. The suffix **r** makes the substitution be as in the ~ command, instead of like *&*.

errorbells, eb default: noeb

Error messages are preceded by a bell.* If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic default: noic

All upper case characters in the text are mapped to lower case in regular expression matching. In addition, all upper case characters in regular expressions are mapped to lower case except in character class specifications.

lisp default: nolisp

Autoindent indents appropriately for *lisp* code, and the **( ) { }** [[and]] commands in *open* and *visual* are modified to have meaning for *lisp*.

list default: nolist

All printed lines will be displayed (more) unambiguously, showing tabs and end-of-lines as in the *list* command.

magic default: magic for *ex* and *vi*†

If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only '↑' and '$' having special effects. In addition the metacharacters '~' and '&' of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a '\'.

mesg default: mesg

Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set.

* Bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb*.

† *Nomagic* for *edit*.

modeline default: nomodeline

If *modeline* is set, then the first 5 lines and the last five lines of the file will be checked for ex command lines and the comands issued. To be recognized as a command line, the line must have the string **ex:** or **vi:** preceeded by a tab or a space. This string may be anywhere in the line and anything after the *:* is interpeted as editor commands. This option defaults to off because of unexpected behavior when editting files such as */etc/passwd*.

number, nu default: nonumber

Causes all output lines to be printed with their line numbers. In addition each input line will be prompted for by supplying the line number it will have.

open default: open

If *noopen*, the commands *open* and *visual* are not permitted. This is set for *edit* to prevent confusion resulting from accidental entry to open or visual mode.

optimize, opt default: optimize

Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para default: para=IPLPPPQPP LIbp

Specifies the paragraphs for the **{** and **}** operations in *open* and *visual*. The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt default: prompt

Command mode input is prompted for with a ':'.

redraw default: noredraw

The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal (e.g. during insertions in *visual* the characters to the right of the cursor position are refreshed as each input character is typed.) Useful only at very high speed.

remap default: remap

If on, macros are repeatedly tried until they are unchanged. For example, if **o** is mapped to **O**, and **O** is mapped to **I**, then if *remap* is set, **o** will map to **I**, but if *noremap* is set, it will map to **O**.

report default: report=5†

Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as *global*, *open*, *undo*, and *visual* which have potentially more far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a *global* command on the individual commands performed.

scroll default: scroll=½ window

Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode *z* command (double the value of *scroll*).

† 2 for *edit*.

sections default: sections=SHNHH HU

Specifies the section macros for the [[and]] operations in *open* and *visual*. The pairs of characters in the options's value are the names of the macros which start paragraphs.

shell, sh default: sh=/bin/sh

Gives the path name of the shell forked for the shell escape command '!', and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw default: sw=8

Gives the width a software tab stop, used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.

showmatch, sm default: nosm

In *open* and *visual* mode, when a) or } is typed, move the cursor to the matching (or { for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow terminal dependent

Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent. See *An Introduction to Display Editing with Vi* for more details.

tabstop, ts default: ts=8

The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl default: tl=0

Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags default: tags=tags /usr/lib/tags

A path of files to be used as tag files for the *tag* command. A requested tag is searched for in the specified files, sequentially. By default, files called **tags** are searched for in the current directory and in /usr/lib (a master file for the entire system).

term from environment TERM

The terminal type of the output device.

terse default: noterse

Shorter error diagnostics are produced for the experienced user.

warn default: warn

Warn if there has been '[No write since last change]' before a '!' command escape.

window default: window=speed dependent

The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600

These are not true options but set **window** only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws default: ws

Searches using the regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm default: wm=0

Defines a margin for automatic wrapover of text during input in *open* and *visual* modes. See *An Introduction to Text Editing with Vi* for details.

writeany, wa default: nowa

Inhibit the checks normally made before *write* commands, allowing a write to any file which the system protection mechanism will allow.

10. Limitations

Editor limits that the user is likely to encounter are as follows: 1024 characters per line, 256 characters per global command list, 128 characters per file name, 128 characters in the previous inserted and deleted text in *open* or *visual*, 100 characters in a shell escape command, 63 characters in a string valued option, and 30 characters in a tag name, and a limit of 250000 lines in the file is silently enforced.

The *visual* implementation limits the number of macros defined with map to 32, and the total number of characters in macros to be less than 512.

Acknowledgments. Chuck Haley contributed greatly to the early development of *ex*. Bruce Englar encouraged the redesign which led to *ex* version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and Unix systems.

SED — A Non-interactive Text Editor

Lee E. McMahon

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed.*

Introduction

Sed is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed.* Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed,* and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

† UNIX is a trademark of AT&T Bell Laboratories.

1. Overall Operation

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

- **-n:** tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);
- **-e:** tells *sed* to take the next argument as an editing command;
- **-f:** tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }')(Sec. 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.
4) The characters '\n' match an imbedded newline character, but not the newline at the end of the pattern space.
5) A period '.' matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets '[ ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression '*d*' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

Examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/\./	matches line 5
/r*an/	matches lines 1,3, 4 (number = zero!)
/\(an\).*\1/	matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after

the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n          n
i\         c\
XXXX       XXXX
d
```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

d (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\(' and '\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by

the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate *w* and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.

A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file 'changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the *g* flag, the command:

```
/X/s/an/AN/p
```

produces (assuming nocopy mode):

```
In XANadu did Kubhla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubhla KhAN
```

3.3. Input-output Functions

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line,

regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a* functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples

Assume that the file 'note1' has the following contents:

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

```
/Kubla/r note1
```

produces:

```
In Xanadu did Kubla Khan
Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson
and most eminent successor of Genghiz (Chingiz) Khan, and founder of the
Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example

The commands

```
1h
1s/ did.*//
1x
G
s/\n/ :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the adress part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

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 Programmer's 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). When you have mastered the Tutorial, try *Advanced Editing on UNIX*. 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. For more on that, read *UNIX for Beginners*.

You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, we will refer to this character, whatever it is, as RETURN.

Getting Started

We'll assume that you have logged in to your system and it has just printed the prompt character, usually either a $ or a %. The easiest way to get *ed* is to type

ed (followed by a return)

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

Creating Text – the Append command "a"

As your first problem, suppose you want to create some text starting from scratch. Perhaps you 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

† UNIX is a trademark of AT&T Bell Laboratories.

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, just type an **a** followed by a RETURN, followed by the lines of text you 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 you 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 you 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 you'll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command

```
w
```

followed by the filename you 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 this case, *ed* would respond with

```
68
```

(Remember that blanks and the return 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 you 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 the prompt character ($ or %). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.†

† Actually, *ed* will print **?** if you try to quit without writing. At that point, write if you want; if not, another **q** will get you out regardless.

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 the prompt character. 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 you 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 you use the **e** command to read a file into the buffer, then you 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 good way to operate is

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

This way, you can simply say **w** from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

You can find out at any time what file name *ed* is remembering by typing the *file* command **f**. In this example, if you typed

```
f
```

ed would reply

```
junk
```

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

Sometimes you 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 you 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 **?name**, where **name** is the name of a file; this means that the file doesn't exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. 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, use the print command

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you 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) say

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

Ed will respond with

Now is the time
for all good men

Suppose you want to print *all* the lines in the buffer. You could use **1,3p** as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you 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, you could use

$,$p

but *ed* lets you abbreviate this to

$p

You 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 you abbreviate even further: you can print any single line by typing *just* the line number – no need to type the letter **p**. So if you say

$

ed will print the last line of the buffer.

You can also use $ in combinations like

$–1,$p

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

Exercise 3:

As before, create some text using the **a** 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 your buffer still contains the six lines as above, that you have just typed

1,3p

and *ed* has printed the three lines for you. 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 you have done anything with. (You just printed it!) You 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 you did anything to (in this case, line 3, which you 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 you most recently did something to." You 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 example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed; the last command will set both . and $ to 6.

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

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

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

.–1 (or .–1p)

which means "print the line *before* the current line." This enables you to go backwards if you 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 return will cause printing of the next line – it's equivalent to **.+1p**. Try it. Try typing a –; you will find that it's equivalent to **.-1p**.

Deleting lines: the "d" command

Suppose you 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 you 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 you 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*. You 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) say

```
p
```

and get

```
Now is the time
```

which is what you 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 you 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, you can try again. (Notice that there is a **p** on the same line as the **s** command. 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 you had

```
Nowxx is the time
```

you can say

```
s/xx//p
```

to get

```
Now is the time
```

Notice that // (two adjacent slashes) 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, you can move on to another highly important idea of *ed* – context searching.

Suppose you have the original three line text in the buffer:

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

Suppose you want to find the line that contains *their* so you 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 you'd been making changes, deleting and rearranging lines, and so on, you 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 to 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* command

```
/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.

You 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**. They were used 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, you could say

/Now/+1s/good/bad/

or

/good/s/good/bad/

or

/party/–1s/good/bad/

The choice is dictated only by convenience. You 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 you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'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 you want to put the first three lines of the buffer at the end instead. You could do it by saying:

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

(Do you see why?) but you 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 you had

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

you could reverse the two paragraphs like this:

/Second/,/end of 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,* **.** means "any character," not a period, so

```
/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,

```
/[0123456789]/
```

matches any single digit – any one of the characters inside the braces will cause a match. This can be abbreviated to **[0–9]**.

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 you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say

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

using your knowledge of **^** and **$**. But the easiest way uses the **&**:

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

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

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

to produce

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

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

```
the end of the world
```

you 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 & saves you from typing it again.

The & is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. You 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, that is, 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 the specified lines to the new text which follows. The new lines are terminated by a ., as with **a**. If no lines are specified, replace line dot. Dot is set to last line changed.

d: 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 new file. Any previous contents of the buffer are thrown away, so issue a **w** beforehand.

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

g: The command

```
g/---/commands
```

will execute the commands on those lines that contain **---**, which can be any context search expression.

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

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

p: Print specified lines. If none specified, print line dot. A single line number is equivalent to *line-number* **p**. A single return prints **.+1**, the next line.

q: Quit *ed*. Wipes out all text in buffer if you give it twice in a row without first giving a **w** command.

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

s: The command

```
s/string1/string2/
```

substitutes the characters **string1** into **string2** in the specified lines. If no lines are specified, make the 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: The command

```
v/---/commands
```

executes **commands** on those lines that *do not* contain **---**.

w: Write out buffer onto a file. Dot is not changed.

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

!: The line

```
!command-line
```

causes **command-line** to be executed as a UNIX command.

/-----/: Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was 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 $.

Part 3

Document Preparation

NROFF/TROFF User's Manual

Joseph F. Ossanna
(updated for 4.3BSD by Mark Seiden)

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the UNIX Time-Sharing System that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. (Device-independent TROFF, part of the Documenter's Workbench, supports additional output devices.) 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*
–i	Read standard input after the input files are exhausted.
–m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files*.
–n*N*	Number first generated page *N*.
–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.
–q	Invoke the simultaneous input-output mode of the **rd** request.
–r*aN*	Number register *a* (one-character) is set to *N*.
–s*N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
–z	Efficiently suppress formatted output. Only produce output to standard error (from **tm** requests or diagnostics).

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.

–h On output, use tabs during horizontal spacing to increase speed. Device tabs setting are assumed to be (and input tabs are initially set to) every 8 character widths.

TROFF Only

–a Send a printable (ASCII) approximation of the results to the standard output.

–b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

–f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

–t Direct output to the standard output instead of the phototypesetter.

–w Wait until phototypesetter is available, if currently busy.

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[1] (for NROFF and TROFF respectively), and the table-construction preprocessor TBL[2]. A reverse-line postprocessor COL[3] 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[3] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TC[5] is a 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* **|** **tc**

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 TC.

The remainder of this manual consists of: a Summary and outline; a Reference Manual keyed to the outline; and a set of Tutorial Examples. Another tutorial is [5].

References

[1] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition),* Bell Laboratories.

[2] M. E. Lesk, *Tbl — A Program to Format Tables,* Bell Laboratories internal memorandum.

[3] Internal on-line documentation (*man* pages) on UNIX.

[4] B. W. Kernighan, *A TROFF Tutorial,* Bell Laboratories.

[5] Your site may have similar programs for more modern displays.

SUMMARY OF REQUESTS AND OUTLINE OF THIS MANUAL

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*.†
.fz *F* ± *N*	off	-	E	font *F* to point size ± *N*.
.fz S *F* ± *N*	off	-	E	Special Font characters to point size ± *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≤ *N*≤ 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.

*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*
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.
.as *xx string*	-	ignored	-	Append *string* to string *xx*.
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	**v**	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	**v**	Change trap location.
.dt *N xx*	-	off	D,**v**	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.
8. Number Registers				
.nr *R ± N M*	-	-	**u**	Define and set number register *R*; auto-increment by *M*.
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=**1**, **i**, **I**, **a**, **A**).
.rr *R*	-	-	-	Remove register *R*.
9. Tabs, Leaders, and Fields				
.ta *Nt* ...	0.8; 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.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
16. Conditional Acceptance of Input				
.if *c anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line 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 $N \leq 0$, 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 error 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.
.ab *string*	-	-	-	Print a message and abort.
.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

ab	20	c2	10	di	7	ex	18	hw	13	lg	10	ne	3	os	5	rd	18	ss	2	uf	10
ad	4	cc	10	ds	7	fc	9	hy	13	li	10	nf	4	pc	14	rm	7	sv	5	ul	10
af	8	ce	4	dt	7	fi	4	ie	16	ll	6	nh	13	pi	19	rn	7	ta	9	vs	5
am	7	ch	7	ec	10	fl	20	if	16	ls	5	nm	15	pl	3	rr	8	tc	9	wh	7
as	7	cs	2	el	16	fp	2	ig	20	lt	14	nn	15	pm	20	rs	5	ti	6		
bd	2	cu	10	em	7	ft	2	in	6	mc	20	nr	8	pn	3	rt	3	tl	14		
bp	3	da	7	eo	10	fz	2	it	7	mk	3	ns	5	po	3	so	19	tm	20		
br	4	de	7	ev	17	hc	13	lc	9	na	4	nx	19	ps	2	sp	5	tr	10		

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 $1 \leq N \leq 9$
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.
19	c.	Number of *lines* read from current input file.
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 (not in ditroff)
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.
5.3	.L	Set to current *line-spacing* (**ls**) parameter
-	.P	Set to 1 if the current page is being printed; otherwise 0.
-	.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 '*.
19	.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.
4.2	.j	Current adjustment mode and type.
4.1	.k	Length of text portion on current partial output line.
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 æsthetic 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 (insertion in place) 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, (for historical reasons, corresponding to the Graphic Systems phototypesetter which had 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 its own internal horizontal/vertical resolution. 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 scale indicator suffixes 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 prefixed to a number N to generate the distance to the vertical or horizontal place N. For vertically-oriented requests and functions, |N becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the vertical place N. For *all* other requests and functions, |N becomes the distance from the current horizontal place on the *input* line to the

horizontal place *N*. For example,

```
.sp |3.2c
```

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected, an expression involving parentheses, the arithmetic operators +, −, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 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 a typesetter-dependent basic character set plus a Special Mathematical Font character set—each having 102 characters. An example of these character sets is shown in the Appendix Table I. All printable ASCII characters are included, with some on the Special Font. With three exceptions, these 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 Appendix 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 are typesetter dependent, but often include 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≤ *N*≤ 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.
.fz *F* ± *N*	off	-	E	The characters in font *F* will be adjusted to be in size ± *N*. Characters in the Special Font encountered during the use of font *F* will have the same size modification. (Use the **.fz S** request if different treatment of Special Font characters is required). **.fz** must follow any **.fp** request for the position.
.fz S *F* ± *N*	off	-	E	The characters in the Special Font will be in size ± *N* independent of previous **.fz** requests.
.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 usable page width on the Graphic Systems phototypesetter was about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper, but these characteristics are typesetter- dependent. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl ± *N*	11 in	11 in	**v**	Page length set to ± *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 ± *N*	*N*=1	-	B*,**v**	Begin page. The current page is ejected and a new page is begun. If ± *N* is given, the new page number will be ± *N*. Also see request **ns**.
.pn ± *N*	*N*=1	ignored	-	Page number. The next page (when it occurs) will have the page number ± *N*. A **pn** must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the **%** register.
.po ± *N*	0; 26/27 in†	previous	**v**	Page offset. The current *left margin* is set to ± *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 distance to the bottom of the page. If $D < V$, another line could still be

*The use of " ' " as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

				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 |\n***R***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 to hyphenate the word 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. The **.k** register (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.

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 (§10.4) can be made to not look like a control line by preceding it by 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. The type indicator can also be a value saved from the read-only **.j** number register, which is set to contain the current adjustment mode and type.

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. The (read-only) number register **.L** is set to contain the current line-spacing value.

				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 outputs a blank line just 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 ($1 \le N \le 9$). 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 the **it** request 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 *Nt* ...	8n; 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 8 character widths. 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 Times Italic on font position 2 (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*' \(space) \0	Move distance *N* Unpaddable space-size space Digit-size space	
\u \d \r	½ em up ½ em down 1 em up	½ line up ½ line down 1 line up	\\| \^	1/6 em space 1/12 em space	ignored 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* are 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 root-en ‾, 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\l'|0\(rn'\l'|0\(ul'
..
```

such that

```
.us "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 $N \leq 0$, 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 $0 \leq N \leq 2$. 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

The (read-only) number register **.c** contains the input line number in the current input file. The number register **c.** is a general register serving the same purpose.

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 occurs immediately. 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. (see §21).
.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.
.ab *string*	-	-	-	Print *string* on standard error and terminate immediately. The default *string* is "User Abort". Does not cause a break. Only output preceding the last break is written.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, **ab** and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard error* 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 much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render

*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).

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, baseline spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg              \"paragraph
.br                 \"break
.ft R               \"force font,
.ps 10              \"size,
.vs 12p             \"spacing,
.in 0               \"and indent
.sp 0.4             \"prespace
.ne 1+\\n(.Vu       \"want more than 1 line
.ti 0.2i            \"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc              \"section
. ---               \"force font, etc.
.sp 0.4             \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1           \"init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp              \"labeled paragraph
.pg
.in 0.5i            \"paragraph indent
.ta 0.2i 0.5i       \"label, paragraph
.ti 0
\t\\$1\t\c          \"flow into paragraph
..
```

The intended usage is "**.lp** *label*"; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with **\c** so that it will become a part of the first line of the text that follows.

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 (all measurements on 8.5 × 11 inch paper prior to photoreduction). This font sample is printed on an APS-5 phototypesetter at University of California, Berkeley.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• □ — - _ ¼ fi fl ff ffi ffl ° † ′ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |*
• □ — - _ ¼ *fi fl ff ffi ffl* ° † ′ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• □ — - _ ¼ 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
κ	\(*k	kappa

Char	*Input Name*	*Character Name*
λ	\(*l	lambda
μ	\(*m	mu
ν	\(*n	nu
ξ	\(*c	xi
ο	\(*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†
Γ	\(*G	Gamma
Δ	\(*D	Delta

Char	*Input Name*	*Character Name*
Ε	\(*E	Epsilon†
Ζ	\(*Z	Zeta†
Η	\(*Y	Eta†
Θ	\(*H	Theta
Ι	\(*I	Iota†
Κ	\(*K	Kappa†
Λ	\(*L	Lambda
Μ	\(*M	Mu†
Ν	\(*N	Nu†
Ξ	\(*C	Xi
Ο	\(*O	Omicron†
Π	\(*P	Pi
Ρ	\(*R	Rho†
Σ	\(*S	Sigma
Τ	\(*T	Tau†
Υ	\(*U	Upsilon
Φ	\(*F	Phi
Χ	\(*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
Ø	\(es	empty set
∈	\(mo	member of
|	\(br	box vertical rule
‡	\(dd	double dagger
☞	\(rh	right hand

Char	*Input Name*	*Character Name*
☜	\(lh	left hand
☮	\(bs	Bell System logo (typesetter-dependent)
|	\(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)

Writing Papers with NROFF using –me

*Eric P. Allman**

Project INGRES
Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes the text processing facilities available on the UNIX† operating system via NROFF† and the –me macro package. It is assumed that the reader already is generally familiar with the UNIX operating system and a text editor such as **ex**. This is intended to be a casual introduction, and as such not all material is covered. In particular, many variations and additional features of the –me macro package are not explained. For a complete discussion of this and other issues, see *The –me Reference Manual* and *The NROFF/TROFF Reference Manual.*

NROFF, a computer program that runs on the UNIX operating system, reads an input file prepared by the user and outputs a formatted paper suitable for publication or framing. The input consists of *text*, or words to be printed, and *requests*, which give instructions to the NROFF program telling how to format the printed copy.

Section 1 describes the basics of text processing. Section 2 describes the basic requests. Section 3 introduces displays. Annotations, such as footnotes, are handled in section 4. The more complex requests which are not discussed in section 2 are covered in section 5. Finally, section 6 discusses things you will need to know if you want to typeset documents. If you are a novice, you probably won't want to read beyond section 4 until you have tried some of the basic features out.

When you have your raw text ready, call the NROFF formatter by typing as a request to the UNIX shell:

nroff –me –T*type files*

where *type* describes the type of terminal you are outputting to. Common values are **dtc** for a DTC 300s (daisy-wheel type) printer and **lpr** for the line printer. If the **–T** flag is omitted, a "lowest common denominator" terminal is assumed; this is good for previewing output on most terminals. A complete description of options to the NROFF command can be found in *The NROFF/TROFF Reference Manual.*

The word *argument* is used in this manual to mean a word or number which appears on the same line as a request which modifies the meaning of that request. For example, the request

.sp

spaces one line, but

.sp 4

spaces four lines. The number **4** is an *argument* to the **.sp** request which says to space four lines instead

*Author's current address: Britton Lee, Inc., 1919 Addison Suite 105, Berkeley, California 94704.

†UNIX is a trademark of AT&T Bell Laboratories

of one. Arguments are separated from the request and from each other by spaces.

1. Basics of Text Processing

The primary function of NROFF is to *collect* words from input lines, *fill* output lines with those words, *justify* the right hand margin by inserting extra spaces in the line, and output the result. For example, the input:

```
Now is the time
for all good men
to come to the aid
of their party.
Four score and seven
years ago,...
```

will be read, packed onto output lines, and justified to produce:

> Now is the time for all good men to come to the aid of their party. Four score and seven years ago,...

Sometimes you may want to start a new output line even though the line you are on is not yet full; for example, at the end of a paragraph. To do this you can cause a *break*, which starts a new output line. Some requests cause a break automatically, as do blank input lines and input lines beginning with a space.

Not all input lines are text to be formatted. Some of the input lines are *requests* which describe how to format the text. Requests always have a period or an apostrophe (" ' ") as the first character of the input line.

The text formatter also does more complex things, such as automatically numbering pages, skipping over page folds, putting footnotes in the correct place, and so forth.

I can offer you a few hints for preparing text for input to NROFF. First, keep the input lines short. Short input lines are easier to edit, and NROFF will pack words onto longer lines for you anyhow. In keeping with this, it is helpful to begin a new line after every period, comma, or phrase, since common corrections are to add or delete sentences or phrases. Second, do not put spaces at the end of lines, since this can sometimes confuse the NROFF processor. Third, do not hyphenate words at the end of lines (except words that should have hyphens in them, such as "mother-in-law"); NROFF is smart enough to hyphenate words for you as needed, but is not smart enough to take hyphens out and join a word back together. Also, words such as "mother-in-law" should not be broken over a line, since then you will get a space where not wanted, such as "mother- in-law".

2. Basic Requests

2.1. Paragraphs

Paragraphs are begun by using the **.pp** request. For example, the input:

```
.pp
Now is the time for all good men
to come to the aid of their party.
Four score and seven years ago,...
```

produces a blank line followed by an indented first line. The result is:

> Now is the time for all good men to come to the aid of their party. Four score and seven years ago,...

Notice that the sentences of the paragraphs *must not* begin with a space, since blank lines and lines beginning with spaces cause a break. For example, if I had typed:

```
.pp
Now is the time for all good men
    to come to the aid of their party.
Four score and seven years ago,...
```

The output would be:

> Now is the time for all good men
> to come to the aid of their party. Four score and seven years ago,...

A new line begins after the word "men" because the second line began with a space character.

There are many fancier types of paragraphs, which will be described later.

2.2. Headers and Footers

Arbitrary headers and footers can be put at the top and bottom of every page. Two requests of the form **.he** *title* and **.fo** *title* define the titles to put at the head and the foot of every page, respectively. The titles are called *three-part* titles, that is, there is a left-justified part, a centered part, and a right-justified part. To separate these three parts the first character of *title* (whatever it may be) is used as a delimiter. Any character may be used, but backslash and double quote marks should be avoided. The percent sign is replaced by the current page number whenever found in the title. For example, the input:

```
.he ''%''
.fo 'Jane Jones''My Book'
```

results in the page number centered at the top of each page, "Jane Jones" in the lower left corner, and "My Book" in the lower right corner.

2.3. Double Spacing

NROFF will double space output text automatically if you use the request **.ls 2**, as is done in this section. You can revert to single spaced mode by typing **.ls 1**.

2.4. Page Layout

A number of requests allow you to change the way the printed copy looks, sometimes called the *layout* of the output page. Most of these requests adjust the placing of "white space" (blank lines or spaces). In these explanations, characters in italics should be replaced with values you wish to use; bold characters represent characters which should actually be typed.

The **.bp** request starts a new page.

The request **.sp** *N* leaves *N* lines of blank space. *N* can be omitted (meaning skip a single line) or can be of the form *N***i** (for *N* inches) or *N***c** (for *N* centimeters). For example, the input:

```
.sp 1.5i
My thoughts on the subject
.sp
```

leaves one and a half inches of space, followed by the line "My thoughts on the subject", followed by a single blank line.

The **.in** +*N* request changes the amount of white space on the left of the page (the *indent*). The argument *N* can be of the form +*N* (meaning leave *N* spaces more than you are already leaving), –*N* (meaning leave less than you do now), or just *N* (meaning leave exactly *N* spaces). *N* can be of the form *N***i** or *N***c** also. For example, the input:

```
initial text
.in 5
some text
.in +1i
more text
.in -2c
final text
```

produces "some text" indented exactly five spaces from the left margin, "more text" indented five spaces plus one inch from the left margin (fifteen spaces on a pica typewriter), and "final text" indented five spaces plus one inch minus two centimeters from the margin. That is, the output is:

```
initial text
          some text
                              more text
     final text
```

The **.ti** *+N* (temporary indent) request is used like **.in** *+N* when the indent should apply to one line only, after which it should revert to the previous indent. For example, the input:

```
.in 1i
.ti 0
Ware, James R.  The Best of Confucius,
Halcyon House, 1950.
An excellent book containing translations of
most of Confucius' most delightful sayings.
A definite must for anyone interested in the early foundations
of Chinese philosophy.
```

produces:

Ware, James R. The Best of Confucius, Halcyon House, 1950. An excellent book containing translations of most of Confucius' most delightful sayings. A definite must for anyone interested in the early foundations of Chinese philosophy.

Text lines can be centered by using the **.ce** request. The line after the **.ce** is centered (horizontally) on the page. To center more than one line, use **.ce** *N* (where *N* is the number of lines to center), followed by the *N* lines. If you want to center many lines but don't want to count them, type:

```
.ce 1000
lines to center
.ce 0
```

The **.ce 0** request tells NROFF to center zero more lines, in other words, stop centering.

All of these requests cause a break; that is, they always start a new line. If you want to start a new line without performing any other action, use **.br**.

2.5. Underlining

Text can be underlined using the **.ul** request. The **.ul** request causes the next input line to be underlined when output. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines (as with the **.ce** request). For example, the input:

```
.ul 2
Notice that these two input lines
are underlined.
```

will underline those eight words in NROFF. (In TROFF they will be set in italics.)

3. Displays

Displays are sections of text to be set off from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this document. All displays except centered blocks are output single spaced.

3.1. Major Quotes

Major quotes are quotes which are several lines long, and hence are set in from the rest of the text without quote marks around them. These can be generated using the commands **.(q** and **.)q** to surround the quote. For example, the input:

```
As Weizenbaum points out:
.(q
It is said that to explain is to explain away.
This maxim is nowhere so well fulfilled
as in the areas of computer programming,...
.)q
```

generates as output:

As Weizenbaum points out:

> It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the areas of computer programming,...

3.2. Lists

A *list* is an indented, single spaced, unfilled display. Lists should be used when the material to be printed should not be filled and justified like normal text, such as columns of figures or the examples used in this paper. Lists are surrounded by the requests **.(l** and **.)l**. For example, type:

```
Alternatives to avoid deadlock are:
.(l
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
.)l
```

will produce:
Alternatives to avoid deadlock are:

```
Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding
```

3.3. Keeps

A *keep* is a display of lines which are kept on a single page if possible. An example of where you would use a keep might be a diagram. Keeps differ from lists in that lists may be broken over a page boundary whereas keeps will not.

Blocks are the basic kind of keep. They begin with the request **.(b** and end with the request **.)b**. If there is not room on the current page for everything in the block, a new page is begun. This has the unpleasant effect of leaving blank space at the bottom of the page. When this is not appropriate, you can use the alternative, called *floating keeps*.

Floating keeps move relative to the text. Hence, they are good for things which will be referred to by name, such as "See figure 3". A floating keep will appear at the bottom of the current page if it will fit; otherwise, it will appear at the top of the next page. Floating keeps begin with the line **.(z** and end with the line **.)z**. For an example of a floating keep, see figure 1.

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1. Example of a Floating Keep.
.hl
.)z
```

Figure 1. Example of a Floating Keep.

The **.hl** request is used to draw a horizontal line so that the figure stands out from the text.

3.4. Fancier Displays

Keeps and lists are normally collected in *nofill* mode, so that they are good for tables and such. If you want a display in fill mode (for text), type **.(l F** (Throughout this section, comments applied to **.(l** also apply to **.(b** and **.(z**). This kind of display will be indented from both margins. For example, the input:

```
.(l F
And now boys and girls,
a newer, bigger, better toy than ever before!
Be the first on your block to have your own computer!
Yes kids, you too can have one of these modern
data processing devices.
You too can produce beautifully formatted papers
without even batting an eye!
.)l
```

will be output as:

> And now boys and girls, a newer, bigger, better toy than ever before! Be the first on your block to have your own computer! Yes kids, you too can have one of these modern data processing devices. You too can produce beautifully formatted papers without even batting an eye!

Lists and blocks are also normally indented (floating keeps are normally left justified). To get a left-justified list, type **.(l L**. To get a list centered line-for-line, type **.(l C**. For example, to get a filled, left justified list, enter:

```
.(l L F
text of block
.)l
```

The input:

```
.(l
first line of unfilled display
more lines
.)l
```

produces the indented text:

```
first line of unfilled display
more lines
```

Typing the character **L** after the **.(l** request produces the left justified result:

first line of unfilled display
more lines

Using **C** instead of **L** produces the line-at-a-time centered output:

first line of unfilled display
more lines

Sometimes it may be that you want to center several lines as a group, rather than centering them one line at a time. To do this use centered blocks, which are surrounded by the requests **.(c** and **.)c**. All the lines are centered as a unit, such that the longest line is centered and the rest are lined up around that line. Notice that lines do not move relative to each other using centered blocks, whereas they do using the **C** argument to keeps.

Centered blocks are *not* keeps, and may be used in conjunction with keeps. For example, to center a group of lines as a unit and keep them on one page, use:

```
.(b L
.(c
first line of unfilled display
more lines
.)c
.)b
```

to produce:

first line of unfilled display
more lines

If the block requests (**.(b** and **.)b**) had been omitted the result would have been the same, but with no guarantee that the lines of the centered block would have all been on one page. Note the use of the **L** argument to **.(b**; this causes the centered block to center within the entire line rather than within the line minus the indent. Also, the center requests must be nested *inside* the keep requests.

4. Annotations

There are a number of requests to save text for later printing. *Footnotes* are printed at the bottom of the current page. *Delayed text* is intended to be a variant form of footnote; the text is printed only when explicitly called for, such as at the end of each chapter. *Indexes* are a type of delayed text having a tag (usually the page number) attached to each entry after a row of dots. Indexes are also saved until called for explicitly.

4.1. Footnotes

Footnotes begin with the request **.(f** and end with the request **.)f**. The current footnote number is maintained automatically, and can be used by typing ******, to produce a footnote number[1]. The number is automatically incremented after every footnote. For example, the input:

[1]Like this.

```
.(q
A man who is not upright
and at the same time is presumptuous;
one who is not diligent and at the same time is ignorant;
one who is untruthful and at the same time is incompetent;
such men I do not count among acquaintances.\**
.(f
\**James R. Ware,
.ul
The Best of Confucius,
Halcyon House, 1950.
Page 77.
.)f
.)q
```

generates the result:

> A man who is not upright and at the same time is presumptuous; one who is not diligent and at the same time is ignorant; one who is untruthful and at the same time is incompetent; such men I do not count among acquaintances.[2]

It is important that the footnote appears *inside* the quote, so that you can be sure that the footnote will appear on the same page as the quote.

4.2. Delayed Text

Delayed text is very similar to a footnote except that it is printed when called for explicitly. This allows a list of references to appear (for example) at the end of each chapter, as is the convention in some disciplines. Use ***#** on delayed text instead of ****** as on footnotes.

If you are using delayed text as your standard reference mechanism, you can still use footnotes, except that you may want to reference them with special characters* rather than numbers.

4.3. Indexes

An "index" (actually more like a table of contents, since the entries are not sorted alphabetically) resembles delayed text, in that it is saved until called for. However, each entry has the page number (or some other tag) appended to the last line of the index entry after a row of dots.

Index entries begin with the request **.(x** and end with **.)x**. The **.)x** request may have a argument, which is the value to print as the "page number". It defaults to the current page number. If the page number given is an underscore ("_") no page number or line of dots is printed at all. To get the line of dots without a page number, type **.)x ""**, which specifies an explicitly null page number.

The **.xp** request prints the index.

For example, the input:

---

[2]James R. Ware, *The Best of Confucius,* Halcyon House, 1950. Page 77.

*Such as an asterisk.

```
.(x
Sealing wax
.)x
.(x
Cabbages and kings
.)x _
.(x
Why the sea is boiling hot
.)x 2.5a
.(x
Whether pigs have wings
.)x ""
.(x
This is a terribly long index entry, such as might be used
for a list of illustrations, tables, or figures; I expect it to
take at least two lines.
.)x
.xp
```

generates:

Sealing wax 9

Cabbages and kings

Why the sea is boiling hot 2.5a

Whether pigs have wings .....

This is a terribly long index entry, such as might be used for a list of illustrations, tables, or figures; I expect it to take at least two lines. 9

The **.(x** request may have a single character argument, specifying the "name" of the index; the normal index is **x**. Thus, several "indices" may be maintained simultaneously (such as a list of tables, table of contents, etc.).

Notice that the index must be printed at the *end* of the paper, rather than at the beginning where it will probably appear (as a table of contents); the pages may have to be physically rearranged after printing.

5. Fancier Features

A large number of fancier requests exist, notably requests to provide other sorts of paragraphs, numbered sections of the form **1.2.3** (such as used in this document), and multicolumn output.

5.1. More Paragraphs

Paragraphs generally start with a blank line and with the first line indented. It is possible to get left-justified block-style paragraphs by using **.lp** instead of **.pp**, as demonstrated by the next paragraph.

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented (opposite of indented) with a label. This can be done with the **.ip** request. A word specified on the same line as **.ip** is printed in the margin, and the body is lined up at a prespecified position (normally five spaces). For example, the input:

```
.ip one
This is the first paragraph.
Notice how the first line
of the resulting paragraph lines up
with the other lines in the paragraph.
.ip two
And here we are at the second paragraph already.
You may notice that the argument to .ip
appears
in the margin.
.lp
We can continue text...
```

produces as output:

one This is the first paragraph. Notice how the first line of the resulting paragraph lines up with the other lines in the paragraph.

two And here we are at the second paragraph already. You may notice that the argument to **.ip** appears in the margin.

We can continue text without starting a new indented paragraph by using the **.lp** request.

If you have spaces in the label of a **.ip** request, you must use an "unpaddable space" instead of a regular space. This is typed as a backslash character ("\") followed by a space. For example, to print the label "Part 1", enter:

```
.ip "Part\ 1"
```

If a label of an indented paragraph (that is, the argument to **.ip**) is longer than the space allocated for the label, **.ip** will begin a new line after the label. For example, the input:

```
.ip longlabel
This paragraph had a long label.
The first character of text on the first line
will not line up with the text on second and subsequent lines,
although they will line up with each other.
```

will produce:

longlabel
This paragraph had a long label. The first character of text on the first line will not line up with the text on second and subsequent lines, although they will line up with each other.

It is possible to change the size of the label by using a second argument which is the size of the label. For example, the above example could be done correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. If you have many paragraphs to indent all the same amount, use the *number register* **ii**. For example, to leave one inch of space for the label, type:

```
.nr ii 1i
```

somewhere before the first call to **.ip**. Refer to the reference manual for more information.

If **.ip** is used with no argument at all no hanging tag will be printed. For example, the input:

```
.ip [a]
This is the first paragraph of the example.
We have seen this sort of example before.
.ip
This paragraph is lined up with the previous paragraph,
but it has no tag in the margin.
```

produces as output:

[a] This is the first paragraph of the example. We have seen this sort of example before.

This paragraph is lined up with the previous paragraph, but it has no tag in the margin.

A special case of **.ip** is **.np**, which automatically numbers paragraphs sequentially from 1. The numbering is reset at the next **.pp**, **.lp**, or **.sh** (to be described in the next section) request. For example, the input:

```
.np
This is the first point.
.np
This is the second point.
Points are just regular paragraphs
which are given sequence numbers automatically
by the .np request.
.pp
This paragraph will reset numbering by .np.
.np
For example,
we have reverted to numbering from one now.
```

generates:

(1) This is the first point.

(2) This is the second point. Points are just regular paragraphs which are given sequence numbers automatically by the .np request.

This paragraph will reset numbering by .np.

(1) For example, we have reverted to numbering from one now.

The **.bu** request gives lists of this sort that are identified with bullets rather than numbers. The paragraphs are also crunched together. For example, the input:

```
.bu
One egg yolk
.bu
One tablespoon cream or top milk
.bu
Salt, cayenne, and lemon juice to taste
.bu
A generous two tablespoonfuls of butter
```

produces[3]:

- One egg yolk

[3]By the way, if you put the first three ingredients in a a heavy, deep pan and whisk the ingredients madly over a medium flame (never taking your hand off the handle of the pot) until the mixture reaches the consistency of custard (just a minute or two), then mix in the butter off-heat, you will have a wonderful Hollandaise sauce.

- One tablespoon cream or top milk
- Salt, cayenne, and lemon juice to taste
- A generous two tablespoonfuls of butter

5.2. Section Headings

Section numbers (such as the ones used in this document) can be automatically generated using the **.sh** request. You must tell **.sh** the *depth* of the section number and a section title. The depth specifies how many numbers are to appear (separated by decimal points) in the section number. For example, the section number **4.2.5** has a depth of three.

Section numbers are incremented in a fairly intuitive fashion. If you add a number (increase the depth), the new number starts out at one. If you subtract section numbers (or keep the same number) the final number is incremented. For example, the input:

```
.sh 1 "The Preprocessor"
.sh 2 "Basic Concepts"
.sh 2 "Control Inputs"
.sh 3
.sh 3
.sh 1 "Code Generation"
.sh 3
```

produces as output the result:

1. The Preprocessor
1.1. Basic Concepts
1.2. Control Inputs
1.2.1.
1.2.2.
2. Code Generation
2.1.1.

You can specify the section number to begin by placing the section number after the section title, using spaces instead of dots. For example, the request:

```
.sh 3 "Another section" 7 3 4
```

will begin the section numbered **7.3.4**; all subsequent **.sh** requests will number relative to this number.

There are more complex features which will cause each section to be indented proportionally to the depth of the section. For example, if you enter:

```
.nr si N
```

each section will be indented by an amount *N*. *N* must have a scaling factor attached, that is, it must be of the form *Nx*, where *x* is a character telling what units *N* is in. Common values for *x* are **i** for inches, **c** for centimeters, and **n** for *ens* (the width of a single character). For example, to indent each section one-half inch, type:

```
.nr si 0.5i
```

After this, sections will be indented by one-half inch per level of depth in the section number. For example, this document was produced using the request

```
.nr si 3n
```

at the beginning of the input file, giving three spaces of indent per section depth.

Section headers without automatically generated numbers can be done using:

```
.uh "Title"
```

which will do a section heading, but will put no number on the section.

5.3. Parts of the Basic Paper

There are some requests which assist in setting up papers. The **.tp** request initializes for a title page. There are no headers or footers on a title page, and unlike other pages you can space down and leave blank space at the top. For example, a typical title page might appear as:

```
.tp
.sp 2i
.(l C
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank N. Furter
.)l
.bp
```

The request **.th** sets up the environment of the NROFF processor to do a thesis, using the rules established at Berkeley. It defines the correct headers and footers (a page number in the upper right hand corner only), sets the margins correctly, and double spaces.

The **.+c** *T* request can be used to start chapters. Each chapter is automatically numbered from one, and a heading is printed at the top of each chapter with the chapter number and the chapter name *T*. For example, to begin a chapter called "Conclusions", use the request:

```
.+c "CONCLUSIONS"
```

which will produce, on a new page, the lines

CHAPTER 5
CONCLUSIONS

with appropriate spacing for a thesis. Also, the header is moved to the foot of the page on the first page of a chapter. Although the **.+c** request was not designed to work only with the **.th** request, it is tuned for the format acceptable for a PhD thesis at Berkeley.

If the title parameter *T* is omitted from the **.+c** request, the result is a chapter with no heading. This can also be used at the beginning of a paper; for example, **.+c** was used to generate page one of this document.

Although papers traditionally have the abstract, table of contents, and so forth at the front of the paper, it is more convenient to format and print them last when using NROFF. This is so that index entries can be collected and then printed for the table of contents (or whatever). At the end of the paper, issue the **.++ P** request, which begins the preliminary part of the paper. After issuing this request, the **.+c** request will begin a preliminary section of the paper. Most notably, this prints the page number restarted from one in lower case Roman numbers. **.+c** may be used repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, etc. The request **.++ B** may also be used to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined in figure 2. (In this figure, comments begin with the sequence \".)

5.4. Equations and Tables

Two special UNIX programs exist to format special types of material. **Eqn** and **neqn** set equations for the phototypesetter and NROFF respectively. **Tbl** arranges to print extremely pretty tables in a variety of formats. This document will only describe the embellishments to the standard features; consult the reference manuals for those processors for a description of their use.

The **eqn** and **neqn** programs are described fully in the document *Typesetting Mathematics – User's Guide* by Brian W. Kernighan and Lorinda L. Cherry. Equations are centered, and are kept

```
.th                          \" set for thesis mode
.fo ''DRAFT''                \" define footer for each page
.tp                          \" begin title page
.(l C                        \" center a large block
THE GROWTH OF TOENAILS
IN UPPER PRIMATES
.sp
by
.sp
Frank Furter
.)l                          \" end centered part
.+c INTRODUCTION             \" begin chapter named "INTRODUCTION"
.(x t                        \" make an entry into index 't'
Introduction
.)x                          \" end of index entry
text of chapter one
.+c "NEXT CHAPTER"           \" begin another chapter
.(x t                        \" enter into index 't' again
Next Chapter
.)x
text of chapter two
.+c CONCLUSIONS
.(x t
Conclusions
.)x
text of chapter three
.++ B                        \" begin bibliographic information
.+c BIBLIOGRAPHY             \" begin another 'chapter'
.(x t
Bibliography
.)x
text of bibliography
.++ P                        \" begin preliminary material
.+c "TABLE OF CONTENTS"
.xp t                        \" print index 't' collected above
.+c PREFACE                  \" begin another preliminary section
text of preface
```

Figure 2. Outline of a Sample Paper

on one page. They are introduced by the **.EQ** request and terminated by the **.EN** request.

The **.EQ** request may take an equation number as an optional argument, which is printed vertically centered on the right hand side of the equation. If the equation becomes too long it should be split between two lines. To do this, type:

```
.EQ (eq 34)
text of equation 34
.EN C
.EQ
continuation of equation 34
.EN
```

The **C** on the **.EN** request specifies that the equation will be continued.

The **tbl** program produces tables. It is fully described (including numerous examples) in the document *Tbl – A Program to Format Tables* by M. E. Lesk. Tables begin with the **.TS** request and end with the **.TE** request. Tables are normally kept on a single page. If you have a table which is too big to fit on a single page, so that you know it will extend to several pages, begin the table with the request **.TS H** and put the request **.TH** after the part of the table which you want duplicated at the top of every page that the table is printed on. For example, a table definition for a long table might look like:

```
.TS H
c s s
n n n.
THE TABLE TITLE
.TH
text of the table
.TE
```

5.5. Two Column Output

You can get two column output automatically by using the request **.2c**. This causes everything after it to be output in two-column form. The request **.bc** will start a new column; it differs from **.bp** in that **.bp** may leave a totally blank column when it starts a new page. To revert to single column output, use **.1c**.

5.6. Defining Macros

A *macro* is a collection of requests and text which may be used by stating a simple request. Macros begin with the line **.de** *xx* (where *xx* is the name of the macro to be defined) and end with the line consisting of two dots. After defining the macro, stating the line *.xx* is the same as stating all the other lines. For example, to define a macro that spaces 3 lines and then centers the next input line, enter:

```
.de SS
.sp 3
.ce
..
```

and use it by typing:

```
.SS
Title Line
(beginning of text)
```

Macro names may be one or two characters. In order to avoid conflicts with names in –me, always use upper case letters as names. The only names to avoid are **TS**, **TH**, **TE**, **EQ**, and **EN**.

5.7. Annotations Inside Keeps

Sometimes you may want to put a footnote or index entry inside a keep. For example, if you want to maintain a "list of figures" you will want to do something like:

```
.(z
.(c
text of figure
.)c
.ce
Figure 5.
.(x f
Figure 5
.)x
.)z
```

which you may hope will give you a figure with a label and an entry in the index **f** (presumably a list of figures index). Unfortunately, the index entry is read and interpreted when the keep is read, not when it is printed, so the page number in the index is likely to be wrong. The solution is to use the magic string **\!** at the beginning of all the lines dealing with the index. In other words, you should use:

```
.(z
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)z
```

which will defer the processing of the index until the figure is output. This will guarantee that the page number in the index is correct. The same comments apply to blocks (with **.(b** and **.)b)** as well.

6. TROFF and the Photosetter

With a little care, you can prepare documents that will print nicely on either a regular terminal or when phototypeset using the TROFF formatting program.

6.1. Fonts

A *font* is a style of type. There are three fonts that are available simultaneously, Times Roman, Times Italic, and Times Bold, plus the special math font. The normal font is Roman. Text which would be underlined in NROFF with the **.ul** request is set in italics in TROFF.

There are ways of switching between fonts. The requests **.r**, **.i**, and **.b** switch to Roman, italic, and bold fonts respectively. You can set a single word in some font by typing (for example):

```
.i word
```

which will set *word* in italics but does not affect the surrounding text. In NROFF, italic and bold text is underlined.

Notice that if you are setting more than one word in whatever font, you must surround that word with double quote marks (‘" ’) so that it will appear to the NROFF processor as a single word. The quote marks will not appear in the formatted text. If you do want a quote mark to appear, you should quote the entire string (even if a single word), and use *two* quote marks where you want one to appear. For example, if you want to produce the text:

"Master Control"

in italics, you must type:

```
.i """Master Control\|"""
```

The \| produces a very narrow space so that the "l" does not overlap the quote sign in TROFF, like this:

"Master Control"

There are also several "pseudo-fonts" available. The input:

```
.(b
.u underlined
.bi "bold italics"
.bx "words in a box"
.)b
```

generates

underlined
bold italics
words in a box

In NROFF these all just underline the text. Notice that pseudo font requests set only the single parameter in the pseudo font; ordinary font requests will begin setting all text in the special font if you do not provide a parameter. No more than one word should appear with these three font requests in the middle of lines. This is because of the way TROFF justifies text. For example, if you were to issue the requests:

```
.bi "some bold italics"
and
.bx "words in a box"
```

in the middle of a line TROFF would produce *some* ***bold*** *italics* and words in a box, which I think you will agree does not look good.

The second parameter of all font requests is set in the original font. For example, the font request:

```
.b bold face
```

generates "bold" in bold font, but sets "face" in the font of the surrounding text, resulting in:

boldface.

To set the two words **bold** and **face** both in **bold face**, type:

```
.b "bold face"
```

You can mix fonts in a word by using the special sequence \c at the end of a line to indicate "continue text processing"; this allows input lines to be joined together without a space between them. For example, the input:

```
.u under \c
.i italics
```

generates under*italics*, but if we had typed:

```
.u under
.i italics
```

the result would have been under *italics* as two words.

6.2. Point Sizes

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text, 8 points for footnotes. To change the pointsize, type:

.sz +*N*

where *N* is the size wanted in points. The *vertical spacing* (distance between the bottom of most letters (the *baseline*) between adjacent lines) is set to be proportional to the type size.

These pointsize changes are *temporary*!!! For example, to reset the pointsize of basic text to twelve point, use:

```
.nr pp 12
.nr sp 12
.nr tp 12
```

to reset the default pointsize of paragraphs, section headers, and titles respectively. If you only want to set the names of sections in a larger pointsize, use:

```
.nr sp 11
```

alone — this sets section titles (e.g., **Point Sizes** above) in a larger font than the default.

A single word or phrase can be set in a smaller pointsize than the surrounding text using the **.sm** request. This is especially convenient for words that are all capitals, due to the optical illusion that makes them look even larger than they actually are. For example:

```
.sm UNIX
```

prints as UNIX rather than UNIX.

Warning: changing point sizes on the phototypesetter is a slow mechanical operation. On laser printers it may require loading new fonts. Size changes should be considered carefully.

6.3. Quotes

It is conventional when using the typesetter to use pairs of grave and acute accents to generate double quotes, rather than the double quote character (‘‘ ’’). This is because it looks better to use grave and acute accents; for example, compare "quote" to ‘‘quote’’.

In order to make quotes compatible between the typesetter and terminals, you may use the sequences ***(lq** and ***(rq** to stand for the left and right quote respectively. These both appear as " on most terminals, but are typeset as ‘‘ and ’’ respectively. For example, use:

```
\*(lqSome things aren't true
even if they did happen.\*(rq
```

to generate the result:

‘‘Some things aren't true even if they did happen.’’

As a shorthand, the special font request:

```
.q "quoted text"
```

will generate ‘‘quoted text’’. Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the –me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; peter kessler for numerous complaints years after I was ‘‘done’’ with this project, most accompanied by fixes (hence forcing me to fix several small bugs); and the plethora of people who have contributed ideas and have given support for the project.

This document was TROFF'ed on September 25, 1991 and applies to version 2.27 of the –me macros.

–ME REFERENCE MANUAL

Release 2.27

*Eric P. Allman**

Project INGRES
Electronics Research Laboratory
University of California, Berkeley
Berkeley, California 94720

This document describes in extremely terse form the features of the **–me** macro package for version seven NROFF/TROFF. Some familiarity is assumed with those programs. Specifically, the reader should understand breaks, fonts, pointsizes, the use and definition of number registers and strings, how to define macros, and scaling factors for ens, points, **v**'s (vertical line spaces), etc.

For a more casual introduction to text processing using NROFF, refer to the document *Writing Papers with NROFF using –me.*

There are a number of macro parameters that may be adjusted. Fonts may be set to a font number only. Font 8 means bold font in TROFF; in NROFF font 8 is underlined unless the **–rb3** flag is specified to use "true bold" font (most versions of NROFF do not interpret bold font nicely). Font 0 is no font change; the font of the surrounding text is used instead. Notice that fonts 0 and 8 are "pseudo-fonts"; that is, they are simulated by the macros. This means that although it is legal to set a font register to zero or eight, it is not legal to use the escape character form, such as:

```
\f8
```

All distances are in basic units, so it is nearly always necessary to use a scaling factor. For example, the request to set the paragraph indent to eight one-en spaces is:

```
.nr pi 8n
```

and not

```
.nr pi 8
```

which would set the paragraph indent to eight basic units, or about 0.02 inch. Default parameter values are given in brackets in the remainder of this document.

Registers and strings of the form $\$x$ may be used in expressions but should not be changed. Macros of the form $\$x$ perform some function (as described) and may be redefined to change this function. This may be a sensitive operation; look at the body of the original macro before changing it.

All names in –me follow a rigid naming convention. The user may define number registers, strings, and macros, provided that s/he uses single character upper case names or double character names consisting of letters and digits, with at least one upper case letter. In no case should special characters be used

*Author's current address: Britton Lee, Inc., 1919 Addison Suite 105, Berkeley, California 94704.

†NROFF and TROFF may be trademarks of AT&T Bell Laboratories.

in user-defined names.

On daisy wheel type printers in twelve pitch, the **–rx1** flag can be stated to make lines default to one eighth inch (the normal spacing for a newline in twelve-pitch). This is normally too small for easy readability, so the default is to space one sixth inch.

The **–rv2** flag will indicates that this *is* being output on a C/A/T phototypesetter; this changes the page offset and inserts cut marks.

This documentation was TROFF'ed on September 24, 1991 and applies to version 2.27 of the –me macros.

1. Paragraphing

These macros are used to begin paragraphs. The standard paragraph macro is **.pp**; the others are all variants to be used for special purposes.

The first call to one of the paragraphing macros defined in this section or the **.sh** macro (defined in the next session) *initializes* the macro processor. After initialization it is not possible to use any of the following requests: **.sc**, **.lo**, **.th**, or **.ac**. Also, the effects of changing parameters which will have a global effect on the format of the page (notably page length and header and footer margins) are not well defined and should be avoided.

.lp Begin left-justified paragraph. Centering and underlining are turned off if they were on, the font is set to **\n(pf** [1] the type size is set to **\n(pp** [10p], and a **\n(ps** space is inserted before the paragraph [0.35v in TROFF, 1v or 0.5v in NROFF depending on device resolution]. The indent is reset to **\n($i** [0] plus **\n(po** [0] unless the paragraph is inside a display. (see **.ba**). At least the first two lines of the paragraph are kept together on a page.

.pp Like **.lp**, except that it puts **\n(pi** [5n] units of indent. This is the standard paragraph macro.

.ip *T I* Indented paragraph with hanging tag. The body of the following paragraph is indented *I* spaces (or **\n(ii** [5n] spaces if *I* is not specified) more than a non-indented paragraph (such as with **.pp**) is. The title *T* is exdented (opposite of indented). The result is a paragraph with an even left edge and *T* printed in the margin. Any spaces in *T* must be unpaddable. If *T* will not fit in the space provided, **.ip** will start a new line.

.np A variant of .ip which numbers paragraphs. Numbering is reset after a **.lp**, **.pp**, or **.sh**. The current paragraph number is in **\n($p**.

.bu Like **.np** except that paragraphs are marked with bullets (•). Leading space is eliminated to create compact lists.

2. Section Headings

Numbered sections are similar to paragraphs except that a section number is automatically generated for each one. The section numbers are of the form **1.2.3**. The *depth* of the section is the count of numbers (separated by decimal points) in the section number.

Unnumbered section headings are similar, except that no number is attached to the heading.

.sh *+N T a b c d e f* Begin numbered section of depth *N*. If *N* is missing the current depth (maintained in the number register **\n($0**) is used. The values of the individual parts of the section number are maintained in **\n($1** through **\n($6**. There is a **\n(ss** [1v] space before the section. *T* is printed as a section title in font **\n(sf** [8] and size **\n(sp** [10p]. The "name" of the section may be accessed via ***($n**. If **\n(si** is non-zero, the base indent is set to **\n(si** times the section depth, and the section title is exdented. (See **.ba**.) Also, an additional indent of **\n(so** [0] is added to the section title (but not to the body of the section). The font is then set to the

paragraph font, so that more information may occur on the line with the section number and title. **.sh** insures that there is enough room to print the section head plus the beginning of a paragraph (about 3 lines total). If *a* through *f* are specified, the section number is set to that number rather than incremented automatically. If any of *a* through *f* are a hyphen that number is not reset. If *T* is a single underscore ("_") then the section depth and numbering is reset, but the base indent is not reset and nothing is printed out. This is useful to automatically coordinate section numbers with chapter numbers.

.sx *+N* — Go to section depth *N* [–1], but do not print the number and title, and do not increment the section number at level *N*. This has the effect of starting a new paragraph at level *N*.

.uh *T* — Unnumbered section heading. The title *T* is printed with the same rules for spacing, font, etc., as for **.sh**.

.$p *T B N* — Print section heading. May be redefined to get fancier headings. *T* is the title passed on the **.sh** or **.uh** line; *B* is the section number for this section, and *N* is the depth of this section. These parameters are not always present; in particular, **.sh** passes all three, **.uh** passes only the first, and **.sx** passes three, but the first two are null strings. Care should be taken if this macro is redefined; it is quite complex and subtle.

.$0 *T B N* — This macro is called automatically after every call to **.$p**. It is normally undefined, but may be used to automatically put every section title into the table of contents or for some similar function. *T* is the section title for the section title which was just printed, *B* is the section number, and *N* is the section depth.

.$1 – .$6 — Traps called just before printing that depth section. May be defined to (for example) give variable spacing before sections. These macros are called from **.$p**, so if you redefine that macro you may lose this feature.

3. Headers and Footers

Headers and footers are put at the top and bottom of every page automatically. They are set in font **\n(tf** [3] and size **\n(tp** [10p]. Each of the definitions apply as of the *next* page. Three-part titles must be quoted if there are two blanks adjacent anywhere in the title or more than eight blanks total.

The spacing of headers and footers are controlled by three number registers. **\n(hm** [4v] is the distance from the top of the page to the top of the header, **\n(fm** [3v] is the distance from the bottom of the page to the bottom of the footer, **\n(tm** [7v] is the distance from the top of the page to the top of the text, and **\n(bm** [6v] is the distance from the bottom of the page to the bottom of the text (nominal). The macros **.m1**, **.m2**, **.m3**, and **.m4** are also supplied for compatibility with ROFF documents.

.he *'l 'm 'r '* — Define three-part header, to be printed on the top of every page.

.fo *'l 'm 'r '* — Define footer, to be printed at the bottom of every page.

.eh *'l 'm 'r '* — Define header, to be printed at the top of every even-numbered page.

.oh *'l 'm 'r '* — Define header, to be printed at the top of every odd-numbered page.

.ef *'l 'm 'r '* — Define footer, to be printed at the bottom of every even-numbered page.

.of *'l 'm 'r '* — Define footer, to be printed at the bottom of every odd-numbered page.

.hx — Suppress headers and footers on the next page.

.m1 *+N* — Set the space between the top of the page and the header [4v].

.m2 *+N* — Set the space between the header and the first line of text [2v].

.m3 *+N* — Set the space between the bottom of the text and the footer [2v].

.m4 *+N* — Set the space between the footer and the bottom of the page [4v].

.ep — End this page, but do not begin the next page. Useful for forcing out footnotes, but other than that hardly every used. Must be followed by a **.bp** or the end of input.

.$h — Called at every page to print the header. May be redefined to provide fancy (e.g., multi-line) headers, but doing so loses the function of the **.he**, **.fo**, **.eh**, **.oh**, **.ef**, and **.of** requests, as well as the chapter-style title feature of **.+c**.

.$f — Print footer; same comments apply as in **.$h**.

.$H — A normally undefined macro which is called at the top of each page (after putting out the header, initial saved floating keeps, etc.); in other words, this macro is called immediately before printing text on a page. It can be used for column headings and the like.

4. Displays

All displays except centered blocks and block quotes are preceded and followed by an extra **\n(bs** [same as **\n(ps**] space. Quote spacing is stored in a separate register; centered blocks have no default initial or trailing space. The vertical spacing of all displays except quotes and centered blocks is stored in register **\n($R** instead of **\n($r**.

.(l *m f* — Begin list. Lists are single spaced, unfilled text. If *f* is **F**, the list will be filled. If *m* [**I**] is **I** the list is indented by **\n(bi** [4m]; if **M** the list is indented to the left margin; if **L** the list is left justified with respect to the text (different from **M** only if the base indent (stored in **\n($i** and set with **.ba**) is not zero); and if **C** the list is centered on a line-by-line basis. The list is set in font **\n(df** [0]. Must be matched by a **.)l**. This macro is almost like **.(b** except that no attempt is made to keep the display on one page.

.)l — End list.

.(q — Begin major quote. These are single spaced, filled, moved in from the text on both sides by **\n(qi** [4n], preceded and followed by **\n(qs** [same as **\n(bs**] space, and are set in point size **\n(qp** [one point smaller than surrounding text].

.)q — End major quote.

.(b *m f* — Begin block. Blocks are a form of *keep*, where the text of a keep is kept together on one page if possible (keeps are useful for tables and figures which should not be broken over a page). If the block will not fit on the current page a new page is begun, *unless* that would leave more than **\n(bt** [0] white space at the bottom of the text. If **\n(bt** is zero, the threshold feature is turned off. Blocks are not filled unless *f* is **F**, when they are filled. The block will be left-justified if *m* is **L**, indented by **\n(bi** [4m] if *m* is **I** or absent, centered (line-for-line) if *m* is **C**, and left justified to the margin (not to the base indent) if *m* is **M**. The block is set in font **\n(df** [0].

.)b — End block.

.(z *m f* — Begin floating keep. Like **.(b** except that the keep is *floated* to the bottom of the page or the top of the next page. Therefore, its position relative to the text changes. The floating keep is preceded and followed by **\n(zs** [1v] space. Also, it defaults to mode **M**.

.)z — End floating keep.

.(c — Begin centered block. The next keep is centered as a block, rather than on a line-by-line basis as with **.(b C**. This call may be nested inside keeps.

.)c End centered block.

5. Annotations

.(d Begin delayed text. Everything in the next keep is saved for output later with **.pd**, in a manner similar to footnotes.

.)d *n* End delayed text. The delayed text number register **\n($d** and the associated string ***#** are incremented if ***#** has been referenced.

.pd Print delayed text. Everything diverted via **.(d** is printed and truncated. This might be used at the end of each chapter.

.(f Begin footnote. The text of the footnote is floated to the bottom of the page and set in font **\n(ff** [1] and size **\n(fp** [8p]. Each entry is preceded by **\n(fs** [0.2v] space, is indented **\n(fi** [3n] on the first line, and is indented **\n(fu** [0] from the right margin. Footnotes line up underneath two column output. If the text of the footnote will not all fit on one page it will be carried over to the next page.

.)f *n* End footnote. The number register **\n($f** and the associated string ****** are incremented if they have been referenced.

.$s The macro to output the footnote separator. This macro may be redefined to give other size lines or other types of separators. Currently it draws a 1.5i line.

.(x *x* Begin index entry. Index entries are saved in the index *x* [**x**] until called up with **.xp**. Each entry is preceded by a **\n(xs** [0.2v] space. Each entry is "undented" by **\n(xu** [0.5i]; this register tells how far the page number extends into the right margin.

.)x *P A* End index entry. The index entry is finished with a row of dots with *A* [null] right justified on the last line (such as for an author's name), followed by P [**\n%**]. If *A* is specified, *P* must be specified; **\n%** can be used to print the current page number. If *P* is an underscore, no page number and no row of dots are printed.

.xp *x* Print index *x* [**x**]. The index is formatted in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

6. Columned Output

.2c *+S N* Enter two-column mode. The column separation is set to *+S* [4n, 0.5i in ACM mode] (saved in **\n($s**). The column width, calculated to fill the single column line length with both columns, is stored in **\n($l**. The current column is in **\n($c**. You can test register **\n($m** [1] to see if you are in single column or double column mode. Actually, the request enters *N* [2] column output.

.1c Revert to single-column mode.

.bc Begin column. This is like **.bp** except that it begins a new column on a new page only if necessary, rather than forcing a whole new page if there is another column left on the current page.

7. Fonts and Sizes

.sz *+P* The pointsize is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to pointsize is stored in **\n($r**. The ratio used internally by displays and annotations is stored in **\n($R** (although this is not used by **.sz**). This size is *not* sticky beyond many macros: in particular, **\n(pp** (paragraph pointsize) modifies the pointsize every time a new paragraph is begun using the **.pp**, **.lp**, **.ip**, **.np**, or **.bu** macros. Also, **\n(fp** (footnote pointsize), **\n(qp** (quote pointsize), **\n(sp** (section header pointsize), and **\n(tp** (title pointsize) may

modify the pointsize.

.r *W X* — Set *W* in roman font, appending *X* in the previous font. To append different font requests, use *X* = **\c**. If no parameters, change to roman font.

.i *W X* — Set *W* in italics, appending *X* in the previous font. If no parameters, change to italic font. Underlines in NROFF.

.b *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. In NROFF, underlines.

.rb *W X* — Set *W* in bold font and append *X* in the previous font. If no parameters, switch to bold font. **.rb** differs from **.b** in that **.rb** does not underline in NROFF.

.u *W X* — Underline *W* and append *X*. This is a true underlining, as opposed to the **.ul** request, which changes to "underline font" (usually italics in TROFF). It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.q *W X* — Quote *W* and append *X*. In NROFF this just surrounds *W* with double quote marks (**‘” ’**), but in TROFF uses directed quotes.

.bi *W X* — Set *W* in bold italics and append *X*. Actually, sets *W* in italic and overstrikes once. Underlines in NROFF. It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

.bx *W X* — Sets *W* in a box, with *X* appended. Underlines in NROFF. It won't work right if *W* is spread or broken (including hyphenated). In other words, it is safe in nofill mode only.

sm *W X* — Sets *W* in a smaller pointsize, with *X* appended.

8. Roff Support

.ix *+N* — Indent, no break. Equivalent to **'in** *N*.

.bl *N* — Leave *N* contiguous white space, on the next page if not enough room on this page. Equivalent to a **.sp** *N* inside a block.

.pa *+N* — Equivalent to **.bp**.

.ro — Set page number in roman numerals. Equivalent to **.af % i**.

.ar — Set page number in Arabic. Equivalent to **.af % 1**.

.n1 — Number lines in margin from one on each page.

.n2 *N* — Number lines from *N*, stop if *N* = 0.

.sk — Leave the next output page blank, except for headers and footers. This is used to leave space for a full-page diagram which is produced externally and pasted in later. To get a partial-page paste-in display, say **.sv** *N*, where *N* is the amount of space to leave; this space will be output immediately if there is room, and will otherwise be output at the top of the next page. However, be warned: if *N* is greater than the amount of available space on an empty page, no space will ever be output.

9. Preprocessor Support

.EQ *m T* — Begin equation. The equation is centered if *m* is **C** or omitted, indented **\n(bi** [4m] if *m* is **I**, and left justified if *m* is **L**. *T* is a title printed on the right margin next to the equation. See *Typesetting Mathematics – User's Guide* by Brian W. Kernighan and Lorinda L. Cherry.

.EN *c* — End equation. If *c* is **C** the equation must be continued by immediately following with another **.EQ**, the text of which can be centered along with this one.

Otherwise, the equation is printed, always on one page, with **\n(es** [0.5v in TROFF, 1v in NROFF] space above and below it.

.TS *h* — Table start. Tables are single spaced and kept on one page if possible. If you have a large table which will not fit on one page, use *h* = **H** and follow the header part (to be printed on every page of the table) with a **.TH**. See *Tbl – A Program to Format Tables* by M. E. Lesk.

.TH — With **.TS H**, ends the header portion of the table.

.TE — Table end. Note that this table does not float, in fact, it is not even guaranteed to stay on one page if you use requests such as **.sp** intermixed with the text of the table. If you want it to float (or if you use requests inside the table), surround the entire table (including the **.TS** and **.TE** requests) with the requests **.(z** and **.)z**.

.PS *h w* — Begin *pic* picture. *H* is the height and *w* is the width, both in basic units. *Ditroff* only.

.PE — End picture.

.IS — Begin *ideal* picture.

.IE — End *ideal* picture.

.IF — End *ideal* picture (alternate form).

GS — Begin *gremlin* picture.

GE — End *gremlin* picture.

GF — End *gremlin* picture (alternate form).

10. Miscellaneous

.re — Reset tabs. Set to every 0.5i in TROFF and every 0.8i in NROFF.

.ba *+N* — Set the base indent to *+N* [0] (saved in **\n($i**). All paragraphs, sections, and displays come out indented by this amount. Titles and footnotes are unaffected. The **.sh** request performs a **.ba** request if **\n(si** [0] is not zero, and sets the base indent to **\n(si*\n($0**.

.xl *+N* — Set the line length to *N* [6.0i]. This differs from **.ll** because it only affects the current environment.

.ll *+N* — Set line length in all environments to *N* [6.0i]. This should not be used after output has begun, and particularly not in two-column output. The current line length is stored in **\n($l**.

.hl — Draws a horizontal line the length of the page. This is useful inside floating keeps to differentiate between the text and the figure.

.lh — Print a letterhead at the current position on the page. The format of the letterhead must be defined in the file **/usr/lib/me/letterhead.me** by your local systems staff. Some environments may require *ditroff* for this macro to function properly.

.lo — This macro loads another set of macros (in **/usr/lib/me/local.me**) which is intended to be a set of locally defined macros. These macros should all be of the form **.*X**, where *X* is any letter (upper or lower case) or digit.

11. Standard Papers

.tp — Begin title page. Spacing at the top of the page can occur, and headers and footers are suppressed. Also, the page number is not incremented for this page.

.th Set thesis mode. This defines the modes acceptable for a doctoral dissertation at Berkeley. It double spaces, defines the header to be a single page number, and changes the margins to be 1.5 inch on the left and one inch on the top. **.++** and **.+c** should be used with it. This macro must be stated before initialization, that is, before the first call of a paragraphing macro or **.sh**.

.++ *m H* This request defines the section of the paper which we are entering. The section type is defined by *m*. **C** means that we are entering the chapter portion of the paper, **A** means that we are entering the appendix portion of the paper, **P** means that the material following should be the preliminary portion (abstract, table of contents, etc.) portion of the paper, **AB** means that we are entering the abstract (numbered independently from 1 in Arabic numerals), and **B** means that we are entering the bibliographic portion at the end of the paper. Also, the variants **RC** and **RA** are allowed, which specify renumbering of pages from one at the beginning of each chapter or appendix, respectively. The *H* parameter defines the new header. If there are any spaces in it, the entire header must be quoted. If you want the header to have the chapter number in it, Use the string **\\\\n(ch**. For example, to number appendixes **A.1** etc., type **.++ RA ''\\\\n(ch.%'**. Each section (chapter, appendix, etc.) should be preceded by the **.+c** request. It should be mentioned that it is easier when using TROFF to put the front material at the end of the paper, so that the table of contents can be collected and put out; this material can then be physically moved to the beginning of the paper.

.+c *T* Begin chapter with title *T*. The chapter number is maintained in **\n(ch**. This register is incremented every time **.+c** is called with a parameter. The title and chapter number are printed by **.$c**. The header is moved to the footer on the first page of each chapter. If *T* is omitted, **.$c** is not called; this is useful for doing your own "title page" at the beginning of papers without a title page proper. **.$c** calls **.$C** as a hook so that chapter titles can be inserted into a table of contents automatically. The footnote numbering is reset to one.

.$c** *T* Print chapter number (from **\n(ch**) and *T*. This macro can be redefined to your liking. It is defined by default to be acceptable for a PhD thesis at Berkeley. This macro calls **$C, which can be defined to make index entries, or whatever.

.$C *K N T* This macro is called by **.$c**. It is normally undefined, but can be used to automatically insert index entries, or whatever. *K* is a keyword, either "Chapter" or "Appendix" (depending on the **.++** mode); *N* is the chapter or appendix number, and *T* is the chapter or appendix title.

.ac *A N* This macro (short for **.acm**) sets up the NROFF environment for camera-ready papers as used by the ACM. This format is 25% larger, and has no headers or footers. The author's name *A* is printed at the bottom of the page (but off the part which will be printed in the conference proceedings), together with the current page number and the total number of pages *N*. Additionally, this macro loads the file **/usr/lib/me/acm.me**, which may later be augmented with other macros useful for printing papers for ACM conferences. It should be noted that this macro will not work correctly in version 7 TROFF, since it sets the page length wider than the physical width of the C/A/T phototypesetter roll.

12. Predefined Strings

****** Footnote number, actually ***[\n(\$f*]**. This macro is incremented after each call to **.)f**.

***#** Delayed text number. Actually **[\n(\$d]**.

*[Superscript. This string gives upward movement and a change to a smaller point size if possible, otherwise it gives the left bracket character ('['). Extra space is left above the line to allow room for the superscript.

*] Unsuperscript. Inverse to *[. For example, to produce a superscript you might type **x*[2*]**, which will produce x^2.

*< Subscript. Defaults to '<' if half-carriage motion not possible. Extra space is left below the line to allow for the subscript.

*> Inverse to *<.

*(dw The day of the week, as a word.

*(mo The month, as a word.

*(td Today's date, directly printable. The date is of the form September 24, 1991. Other forms of the date can be used by using **\n(dy** (the day of the month; for example, 24), ***(mo** (as noted above) or **\n(mo** (the same, but as an ordinal number; for example, September is 9), and **\n(yr** (the last two digits of the current year).

*(lq Left quote marks. Double quote in NROFF.

*(rq Right quote.

*– ¾ em dash in TROFF; two hyphens in NROFF.

13. Special Characters and Marks

There are a number of special characters and diacritical marks (such as accents) available through –me. To reference these characters, you must call the macro **.sc** to define the characters before using them.

.sc Define special characters and diacritical marks, as described in the remainder of this section. This macro must be stated before initialization. The special characters available are listed below.

Name	Usage	Example	
Acute accent	*´	a*´	á
Grave accent	*`	e*`	è
Umlat	*:	u*:	ü
Tilde	*~	n*~	ñ
Caret	*^	e*^	ê
Cedilla	*,	c*,	ç
Czech	*v	e*v	ě
Circle	*o	A*o	Å
There exists	*(qe		∃
For all	*(qa		∀

Acknowledgments

I would like to thank Bob Epstein, Bill Joy, and Larry Rowe for having the courage to use the –me macros to produce non-trivial papers during the development stages; Ricki Blau, Pamela Humphrey, and Jim Joyce for their help with the documentation phase; peter kessler for numerous complaints, most accompanied by fixes; and the plethora of people who have contributed ideas and have given support for the project.

Summary

This alphabetical list summarizes all macros, strings, and number registers available in the –me macros. Selected *troff* commands, registers, and functions are included as well; those listed can generally be used with impunity.

The columns are the name of the command, macro, register, or string; the type of the object, and the description. Types are **M** for macro or builtin command (invoked with **.** or ´ in the first input column), **S** for a string (invoked with ***** or ***(**), **R** for a number register (invoked with **\n** or **\n(**), and **F** for a *troff* builtin function (invoked by preceding it with a single backslash).

Lines marked with § are *troff* internal codes. Lines marked with † or ‡ may be defined by the user to get special functions; ‡ indicates that these are defined by default and changing them may have unexpected side effects. Lines marked with ° are specific to *ditroff* (device-independent *troff*).

NAME	TYPE	DESCRIPTION
\(space)	F§	unpaddable space
\"	F§	comment (to end of line)
*#	S	optional delayed text tag string
\$*N*	F§	interpolate argument *N*
\n($0	R	section depth
.$0	M†	invoked after section title printed
\n($1	R	first section number
.$1	M†	invoked before printing depth 1 section
\n($2	R	second section number
.$2	M†	invoked before printing depth 2 section
\n($3	R	third section number
.$3	M†	invoked before printing depth 3 section
\n($4	R	fourth section number
.$4	M†	invoked before printing depth 4 section
\n($5	R	fifth section number
.$5	M†	invoked before printing depth 5 section
\n($6	R	sixth section number
.$6	M†	invoked before printing depth 6 section
.$C	M†	called at beginning of chapter
.$H	M†	text header
\n($R	R‡	relative vertical spacing in displays
\n($c	R	current column number
.$c	M‡	print chapter title
\n($d	R	delayed text number
\n($f	R	footnote number
.$f	M‡	print footer
.$h	M‡	print header
\n($i	R	paragraph base indent
\n($l	R	column width
\n($m	R	number of columns in effect
*($n	S	section name
\n($p	R	numbered paragraph number
.$p	M‡	print section heading (internal macro)
\n($r	R‡	relative vertical spacing in text
\n($s	R	column indent
.$s	M‡	footnote separator (from text)
\n%	R§	current page number
\&	F§	zero width character, useful for hiding controls
\(*xx*	F§	interpolate special character *xx*
.(b	M	begin block

NAME	TYPE	DESCRIPTION
.(c	M	begin centered block
.(d	M	begin delayed text
.(f	M	begin footnote
.(l	M	begin list
.(q	M	begin quote
.(x	M	begin index entry
.(z	M	begin floating keep
.)b	M	end block
.)c	M	end centered block
.)d	M	end delayed text
.)f	M	end footnote
.)l	M	end list
.)q	M	end quote
.)x	M	end index entry
.)z	M	end floating keep
*x	F§	interpolate string *x*
*(xx	F§	interpolate string *xx*
**	S	optional footnote tag string
.++	M	set paper section type
.+c	M	begin chapter
*,	S	cedilla
\–	F§	minus sign
*–	S	3/4 em dash
\0	F§	unpaddable digit-width space
.1c	M	revert to single column output
.2c	M	begin two column output
*:	S	umlat
*<	S	begin subscript
*>	S	end subscript
.EN	M	end equation
.EQ	M	begin equation
\L'*d*'	F§	vertical line drawing function for distance *d*
.GE	M•	end *gremlin* picture
.GF	M•	end *gremlin* picture (with flyback)
.GS	M•	start *gremlin* picture
.IE	M•	end *ideal* picture
.IF	M•	end *ideal* picture (with flyback)
.IS	M•	start *ideal* picture
.PE	M•	end *pic* picture
.PF	M•	end *pic* picture (with flyback)
.PS	M•	start *pic* picture
.TE	M	end table
.TH	M	end header of table
.TS	M	begin table
*[	S	begin superscript
\n(.$	R§	number of arguments to macro
\n(.i	R§	current indent
\n(.l	R§	current line length
\n(.s	R§	current point size
*(´	S	acute accent
*(`	S	grave accent
\(´	F§	acute accent

NAME	TYPE	DESCRIPTION
\(`	F§	grave accent
*]	S	end superscript
\^	F§	1/12 em narrow space
*^	S	caret
.ac	M	ACM mode
.ad	M§	set text adjustment
.af	M§	assign format to register
.am	M§	append to macro
.ar	M	set page numbers in Arabic
.as	M§	append to string
.b	M	bold font
.ba	M	set base indent
.bc	M	begin new column
.bi	M	bold italic
\n(bi	R	display (block) indent
.bl	M	blank lines (even at top of page)
\n(bm	R	bottom title margin
.bp	M§	begin page
.br	M§	break (start new line)
\n(bs	R	display (block) pre/post spacing
\n(bt	R	block keep threshold
.bx	M	boxed
\c	F§	continue input
.ce	M§	center lines
\n(ch	R	current chapter number
.de	M§	define macro
\n(df	R	display font
.ds	M§	define string
\n(dw	R§	current day of week
*(dw	S	current day of week
\n(dy	R§	day of month
\e	F§	printable version of \
.ef	M	set footer (even numbered pages only)
.eh	M	set header (even numbered pages only)
.el	M§	else part of conditional
.ep	M	end page
\n(es	R	equation pre/post space
\f*f*	F§	inline font change to font *f*
\f(*ff*	F§	inline font change to font *ff*
.fc	M§	set field characters
\n(ff	R	footnote font
.fi	M§	fill output lines
\n(fi	R	footnote indent (first line only)
\n(fm	R	footer margin
.fo	M	set footer
\n(fp	R	footnote pointsize
\n(fs	R	footnote prespace
\n(fu	R	footnote undent (from right margin)
\h'*d*'	F§	local horizontal motion for distance *d*
.hc	M§	set hyphenation character
.he	M	set header
.hl	M	draw horizontal line

NAME	TYPE	DESCRIPTION
\n(hm	R	header margin
.hx	M	suppress headers and footers on next page
.hy	M§	set hyphenation mode
.i	M	italic font
.ie	M§	conditional with else
.if	M§	conditional
\n(ii	R	indented paragraph indent
.in	M§	indent (transient, use .ba for pervasive)
.ip	M	begin indented paragraph
.ix	M	indent, no break
\l'*d*'	F§	horizontal line drawing function for distance *d*
.lc	M§	set leader repetition character
.lh	M•	interpolate local letterhead
.ll	M	set line length
.lo	M	load local macros
.lp	M	begin left justified paragraph
*(lq	S	left quote marks
.ls	M§	set multi-line spacing
.m1	M	set space from top of page to header
.m2	M	set space from header to text
.m3	M	set space from text to footer
.m4	M	set space from footer to bottom of page
.mc	M§	insert margin character
.mk	M§	mark vertical position
\n(mo	R§	month of year
*(mo	S	current month
\n*x*	F§	interpolate number register *x*
\n(*xx*	F§	interpolate number register *xx*
.n1	M	number lines in margin
.n2	M	number lines in margin
.na	M§	turn off text adjustment
.ne	M§	need vertical space
.nf	M§	don't fill output lines
.nh	M§	turn off hyphenation
.np	M	begin numbered paragraph
.nr	M§	set number register
.ns	M§	no space mode
*o	S	circle (e.g., for Norse Å)
.of	M	set footer (odd numbered pages only)
.oh	M	set header (odd numbered pages only)
.pa	M	begin page
.pd	M	print delayed text
\n(pf	R	paragraph font
\n(pi	R	paragraph indent
.pl	M§	set page length
.pn	M§	set next page number
.po	M§	page offset
\n(po	R	simulated page offset
.pp	M	begin paragraph
\n(pp	R	paragraph pointsize
\n(ps	R	paragraph prespace
.q	M	quoted

NAME	TYPE	DESCRIPTION
*(qa	S	for all
*(qe	S	there exists
\n(qi	R	quote indent (also shortens line)
\n(qp	R	quote pointsize
\n(qs	R	quote pre/post space
.r	M	roman font
.rb	M	real bold font
.re	M	reset tabs
.rm	M§	remove macro or string
.rn	M§	rename macro or string
.ro	M	set page numbers in roman
*(rq	S	right quote marks
.rr	M§	remove register
.rs	M§	restore spacing
.rt	M§	return to vertical position
\s*S*	F§	inline size change to size *S*
.sc	M	load special characters
\n(sf	R	section title font
.sh	M	begin numbered section
\n(si	R	relative base indent per section depth
.sk	M	skip next page
.sm	M	set argument in a smaller pointsize
.so	M§	source input file
\n(so	R	additional section title offset
.sp	M§	vertical space
\n(sp	R	section title pointsize
\n(ss	R	section prespace
.sx	M	change section depth
.sz	M	set pointsize and vertical spacing
.ta	M§	set tab stops
.tc	M§	set tab repetition character
*(td	S	today's date
\n(tf	R	title font
.th	M	set thesis mode
.ti	M§	temporary indent (next line only)
.tl	M§	three part title
\n(tm	R	top title margin
.tp	M	begin title page
\n(tp	R	title pointsize
.tr	M§	translate
.u	M	underlined
.uh	M	unnumbered section
.ul	M§	underline next line
\v'*d*'	F§	local vertical motion for distance *d*
*v	S	inverted 'v' for czeck "$\overset{v}{e}$"
\w'*S*'	F§	return width of string *S*
.xl	M	set line length (local)
.xp	M	print index
\n(xs	R	index entry prespace
\n(xu	R	index undent (from right margin)
\n(yr	R§	year (last two digits only)
\n(zs	R	floating keep pre/post space

NAME	TYPE	DESCRIPTION
\{	F§	begin conditional group
\\|	F§	1/6 em narrow space
\}	F§	end conditional group
*~	S	tilde

Typing Documents on the UNIX System: Using the –ms Macros with Troff and Nroff

M. E. Lesk

ABSTRACT

This document describes a set of easy-to-use macros for preparing documents on the UNIX system. Documents may be produced on either the phototypesetter or a on a computer terminal, without changing the input.

The macros provide facilities for paragraphs, sections (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, and cover pages for papers.

This memo includes, as an appendix, the text of the "Guide to Preparing Documents with –ms" which contains additional examples of features of –ms.

This manual is a revision of, and replaces, "Typing Documents on UNIX," dated November 22, 1974.

Introduction. This memorandum describes a package of commands to produce papers using the *troff* and *nroff* formatting programs on the UNIX system. As with other *roff*-derived programs, text is prepared interspersed with formatting commands. However, this package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. The commands available in this package are listed in Appendix A.

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

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

Beginning. For a document with a paper-type cover sheet, the input should start as follows:

```
[optional overall format .RP – see below]
.TL
Title of document (one or more lines)
.AU
Author(s) (may also be several lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.AE  (abstract end)
text ... (begins with .PP, which see)
```

To omit some of the standard headings (e.g. no abstract, or no author's institution) just omit the corresponding fields and command lines. The word ABSTRACT can be suppressed by writing ".AB no" for ".AB". Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory: beginning with a .PP command is perfectly OK and will just start printing an ordinary paragraph. ***Warning:*** You can't just begin a document with a line of text.

Some –ms command must precede any text input. When in doubt, use .LP to get proper initialization, although any of the commands .PP, .LP, .TL, .SH, .NH is good enough. Figure 1 shows the legal arrangement of commands at the start of a document.

Cover Sheets and First Pages. The first line of a document signals the general format of the first page. In particular, if it is ".RP" a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

In general –ms is arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and unnecessary items for that format are ignored.

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

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

Multi-column formats. If you place the command ".2C" in your document, the document will be printed in double column format beginning at that point. This feature is not too useful in computer terminal output, but is often desirable on the typesetter. The command ".1C" will go back to one-column format and also skip to a new page. The ".2C" command is actually a special case of the command

.MC [column width [gutter width]]

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

Headings. To produce a special heading, there are two commands. If you type

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

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

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

produces

1. Care and Feeding of Department Heads

Alternatively,

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

will print the heading with no number added:

Care and Feeding of Directors

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

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

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

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

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

```
.NH 0
Penn Central
```

1. Penn Central

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

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

produces

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

[2] Text for second paragraph, ...

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

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

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

will produce

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

If a non-standard amount of indenting is required, it may be specified after the label (in character positions) and will remain in effect until the next .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

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

produces this:

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

second: And so forth.

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

```
.IP 1.
Bell Laboratories
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

will result in

1. Bell Laboratories
   - 1.1 Murray Hill
   - 1.2 Holmdel
   - 1.3 Whippany
     - 1.3.1 Madison
   - 1.4 Chester

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

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

Emphasis. To get italics (on the typesetter) or underlining (on the terminal) say

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

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

```
.I word
```

and in this case no .R is needed to restore the previous font. **Boldface** can be produced by

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

and also will be underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on the same line as the .B command.

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

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

```
.UL word
```

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

Footnotes. Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page*. By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below).

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

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

By default, lines between .DS and .DE are indented and left-adjusted. You can also center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered (and not re-arranged); lines bracketed by .DS L and .DE are left-adjusted, not indented, and not re-arranged. A plain .DS is equivalent to .DS I, which indents and left-adjusts. Thus,

* Like this.

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

whereas

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

Note that .DS C centers each line; there is a variant .DS B that makes the display into a left-adjusted block of text, and then centers that entire block. Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I respectively. An extra argument to the .DS I or .DS command is taken as an amount to indent. Note: it is tempting to assume that .DS R will right adjust lines, but it doesn't work.

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

.BX word

will print word as shown. The boxes will not be neat on a terminal, and this should not be used as a substitute for italics.

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

.B1
text...
.B2

as has been done here.

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

Nroff/Troff commands. Among the useful commands from the basic formatting programs are the following. They all work with both typesetter and computer terminal output:

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

Date. By default, documents produced on computer terminals have the date at the bottom of each page; documents produced on the typesetter don't. To force the date, say ".DA". To force no date, say ".ND". To lie about the date, say ".DA July 4, 1776" which puts the specified date at the bottom of each page. The command

.ND May 8, 1945

in ".RP" format places the specified date on the cover sheet and nowhere else. Place this line before the title.

Signature line. You can obtain a signature line by placing the command .SG in the document. The authors' names will be output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

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

.nr PS 9

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

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6″
LT	title length	next para.	6″
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27″
HM	top margin	next page	1″
FM	bottom margin	next page	1″

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and

RF which are strings in the page footer. The page number on *output* is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

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

Input	Output	Input	Output
*'e	é	*~a	ã
*`e	è	*Ce	e̊
*:u	ü	*,c	ç
*^e	ê		

Use. After your document is prepared and stored on a file, you can print it on a terminal with the command*

nroff –ms file

and you can print it on the typesetter with the command

troff –ms file

(many options are possible). In each case, if your document is stored in several files, just list all the filenames where we have used "file". If equations or tables are used, *eqn* and/or *tbl* must be invoked as preprocessors.

References and further study. If you have to do Greek or mathematics, see ***eqn*** [1] for equation setting. To aid ***eqn*** users, ***–ms*** provides definitions of .EQ and .EN which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

.EQ L (1.3a)

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

Similarly, the macros .TS and .TE are defined to separate tables (see [2]) from text with a little space. A very long table with a heading may be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data after the heading. If the table has no heading repeated from page to page, just use the ordinary .TS and .TE macros.

To learn more about *troff* see [3] for a general introduction, and [4] for the full details (experts only). Information on related UNIX commands is in [5]. For jobs that do not seem well-adapted to –ms, consider other macro packages. It is often far easier to write a specific macro packages for such tasks as imitating particular journals than to try to adapt –ms.

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

References

[1] B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics — Users Guide (2nd edition),* Bell Laboratories Computing Science Report no. 17.

[2] M. E. Lesk, *Tbl — A Program to Format Tables,* Bell Laboratories Computing Science Report no. 45.

[3] B. W. Kernighan, *A Troff Tutorial,* Bell Laboratories, 1976.

[4] J. F. Ossanna, *Nroff/Troff Reference Manual,* Bell Laboratories Computing Science Report no. 51.

[5] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, 1978.

* If .2C was used, pipe the *nroff* output through *col;* make the first line of the input ".pi /usr/bin/col."

Appendix A
List of Commands

1C	Return to single column format.
2C	Start double column format.
AB	Begin abstract.
AE	End abstract.
AI	Specify author's institution.
AU	Specify author.
B	Begin boldface.
DA	Provide the date on each page.
DE	End display.
DS	Start display (also CD, LD, ID).
EN	End equation.
EQ	Begin equation.
FE	End footnote.
FS	Begin footnote.
I	Begin italics.
IP	Begin indented paragraph.
KE	Release keep.
KF	Begin floating keep.
KS	Start keep.
LG	Increase type size.
LP	Left aligned block paragraph.
ND	Change or cancel date.
NH	Specify numbered heading.
NL	Return to normal type size.
PP	Begin paragraph.
R	Return to regular font (usually Roman).
RE	End one level of relative indenting.
RP	Use released paper format.
RS	Relative indent increased one level.
SG	Insert signature line.
SH	Specify section heading.
SM	Change to smaller type size.
TL	Specify title.
UL	Underline one word.

Register Names

The following register names are used by –ms internally. Independent use of these names in one's own macros may produce incorrect output. Note that no lower case letters are used in any –ms internal name.

Number registers used in –ms

:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
1T	FL	H3	IK	KI	MM	NF	PF	PX	TD	YE
AV	FM	H4	IM	L1	MN	NS	PI	RO	TN	YY
CW	FP	H5	IP	LE	MO	OI	PN	ST	TQ	ZN

String registers used in –ms

'	A5	CB	DW	EZ	I	KF	MR	R1	RT	TL
`	AB	CC	DY	FA	I1	KQ	ND	R2	S0	TM
^	AE	CD	E1	FE	I2	KS	NH	R3	S1	TQ
~	AI	CF	E2	FJ	I3	LB	NL	R4	S2	TS
:	AU	CH	E3	FK	I4	LD	NP	R5	SG	TT
,	B	CM	E4	FN	I5	LG	OD	RC	SH	UL
1C	BG	CS	E5	FO	ID	LP	OK	RE	SM	WB
2C	BT	CT	EE	FQ	IE	ME	PP	RF	SN	WH
A1	C	D	EL	FS	IM	MF	PT	RH	SY	WT
A2	C1	DA	EM	FV	IP	MH	PY	RP	TA	XD
A3	C2	DE	EN	FY	IZ	MN	QF	RQ	TE	XF
A4	CA	DS	EQ	HO	KE	MO	R	RS	TH	XK

Order of Commands in Input

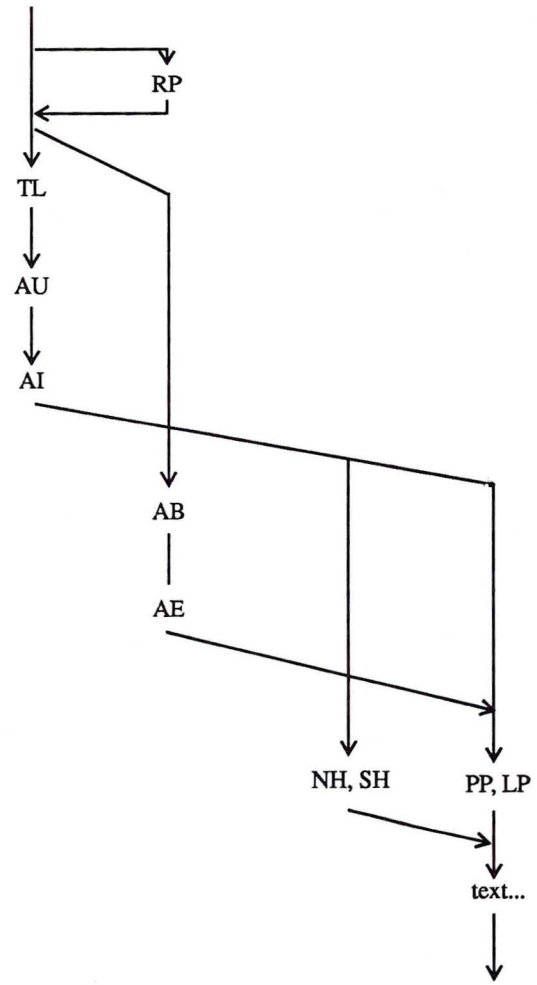


Figure 1

A Revised Version of -ms

Bill Tuthill

Computing Services
University of California
Berkeley, CA 94720

The -ms macros have been slightly revised and rearranged for the Berkeley Unix distribution. Because of the rearrangement, the new macros can be read by the computer in about half the time required by the previous version of -ms. This means that output will begin to appear between ten seconds and several minutes more quickly, depending on the system load. On long files, however, the savings in total time are not substantial. The old version of -ms is still available as -mos.

Several bugs in -ms have been fixed, including a bad problem with the .1C macro, minor difficulties with boxed text, a break induced by .EQ before initialization, the failure to set tab stops in displays, and several bothersome errors in the **refer** macros. Macros used only at Bell Laboratories have been removed. There are a few extensions to previous -ms macros, and a number of new macros, but all the documented -ms macros still work exactly as they did before, and have the same names as before. Output produced with -ms should look like output produced with -mos.

One important new feature is automatically numbered footnotes. Footnote numbers are printed by means of a pre-defined string (**), which you invoke separately from .FS and .FE. Each time it is used, this string increases the footnote number by one, whether or not you use .FS and .FE in your text. Footnote numbers will be superscripted on the phototypesetter and on daisy-wheel terminals, but on low-resolution devices (such as the lpr and a crt), they will be bracketed. If you use ** to indicate numbered footnotes, then the .FS macro will automatically include the footnote number at the bottom of the page. This footnote, for example, was produced as follows:[1]

```
This footnote, for example, was produced as follows:\**
.FS
    ...
.FE
```

If you are using ** to number footnotes, but want a particular footnote to be marked with an asterisk or a dagger, then give that mark as the first argument to .FS: †

```
then give that mark as the first argument to .FS: \(dg
.FS  \(dg
    ...
.FE
```

Footnote numbering will be temporarily suspended, because the ** string is not used. Instead of a dagger, you could use an asterisk * or double dagger ‡, represented as \(dd.

Another new feature is a macro for printing theses according to Berkeley standards. This macro is called .TM, which stands for thesis mode. (It is much like the .th macro in -me.) It will put page numbers in the upper right-hand corner; number the first page; suppress the date; and doublespace everything except quotes, displays, and keeps. Use it at the top of each file making up your thesis. Calling

[1] If you never use the "**" string, no footnote numbers will appear anywhere in the text, including down here. The output footnotes will look exactly like footnotes produced with -mos.

† In the footnote, the dagger will appear where the footnote number would otherwise appear, as on the left.

.TM defines the .CT macro for chapter titles, which skips to a new page and moves the pagenumber to the center footer. The .P1 (P one) macro can be used even without thesis mode to print the header on page 1, which is suppressed except in thesis mode. If you want roman numeral page numbering, use an ".af PN i" request.

There is a new macro especially for bibliography entries, called .XP, which stands for exdented paragraph. It will exdent the first line of the paragraph by \n(PI units, usually 5n (the same as the indent for the first line of a .PP). Most bibliographies are printed this way. Here are some examples of exdented paragraphs:

Lumley, Lyle S., *Sex in Crustaceans: Shell Fish Habits,* Harbinger Press, Tampa Bay and San Diego, October 1979. 243 pages. The pioneering work in this field.

Leffadinger, Harry A., "Mollusk Mating Season: 52 Weeks, or All Year?" in *Acta Biologica,* vol. 42, no. 11, November 1980. A provocative thesis, but the conclusions are wrong.

Of course, you will have to take care of italicizing the book title and journal, and quoting the title of the journal article. Indentation or exdentation can be changed by setting the value of number register PI.

If you need to produce endnotes rather than footnotes, put the references in a file of their own. This is similar to what you would do if you were typing the paper on a conventional typewriter. Note that you can use automatic footnote numbering without actually having .FS and .FE pairs in your text. If you place footnotes in a separate file, you can use .IP macros with ** as a hanging tag; this will give you numbers at the left-hand margin. With some styles of endnotes, you would want to use .PP rather then .IP macros, and specify ** before the reference begins.

There are four new macros to help produce a table of contents. Table of contents entries must be enclosed in .XS and .XE pairs, with optional .XA macros for additional entries; arguments to .XS and .XA specify the page number, to be printed at the right. A final .PX macro prints out the table of contents. Here is a sample of typical input and output text:

```
.XS ii
Introduction
.XA 1
Chapter 1: Review of the Literature
.XA 23
Chapter 2: Experimental Evidence
.XE
.PX
```

Table of Contents

The .XS and .XE pairs may also be used in the text, after a section header for instance, in which case page numbers are supplied automatically. However, most documents that require a table of contents are too long to produce in one run, which is necessary if this method is to work. It is recommended that you do a table of contents after finishing your document. To print out the table of contents, use the .PX macro; if you forget it, nothing will happen.

As an aid in producing text that will format correctly with both **nroff** and **troff**, there are some new string definitions that define quotation marks and dashes for each of these two formatting programs. The *- string will yield two hyphens in **nroff**, but in **troff** it will produce an em dash— like this one. The *Q and *U strings will produce " and " in **troff**, but " in **nroff**. (In typesetting, the double quote is traditionally considered bad form.)

There are now a large number of optional foreign accent marks defined by the -ms macros. All the accent marks available in -mos are present, and they all work just as they always did. However, there are better definitions available by placing .AM at the beginning of your document. Unlike the -mos accent marks, the accent strings should come *after* the letter being accented. Here is a list of the diacritical marks, with examples of what they look like.

name of accent	input	output
acute accent	e*'	é
grave accent	e*`	è
circumflex	o*^	ô
cedilla	c*,	ç
tilde	n*~	ñ
question	*?	¿
exclamation	*!	¡
umlaut	u*:	ü
digraph s	*8	ß
hacek	c*v	č
macron	a*_	ā
underdot	s*.	ṣ
o-slash	o*/	ǫ́
angstrom	a*o	å
yogh	kni*3t	kniȝt
Thorn	*(Th	Þ
thorn	*(th	þ
Eth	*(D-	Ð
eth	*(d-	ð
hooked o	*q	ǫ
ae ligature	*(ae	æ
AE ligature	*(Ae	Æ
oe ligature	*(oe	œ
OE ligature	*(Oe	Œ

If you want to use these new diacritical marks, don't forget the .AM at the top of your file. Without it, some will not print at all, and others will be placed on the wrong letter.

It is also possible to produce custom headers and footers that are different on even and odd pages. The .OH and .EH macros define odd and even headers, while .OF and .EF define odd and even footers. Arguments to these four macros are specified as with .tl. This document was produced with:

```
.OH '\fIThe -mx Macros''Page %\fP'
.EH '\fIPage %''The -mx Macros\fP'
```

Note that it would be a error to have an apostrophe in the header text; if you need one, you will have to use a different delimiter around the left, center, and right portions of the title. You can use any character as a delimiter, provided it doesn't appear elsewhere in the argument to .OH, .EH, .OF, or EF.

The -ms macros work in conjunction with the **tbl**, **eqn**, and **refer** preprocessors. Macros to deal with these items are read in only as needed, as are the thesis macros (.TM), the special accent mark definitions (.AM), table of contents macros (.XS and .XE), and macros to format the optional cover page. The code for the -ms package lives in /usr/lib/tmac/tmac.s, and sourced files reside in the directory /usr/ucb/lib/ms.

Tbl — A Program to Format Tables

M. E. Lesk

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 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

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 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])

† UNIX is a trademark of AT&T Bell Laboratories.

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.
   linesize (n) — set lines or rules (e.g. from **box**) in n point type;
   delim (xy) — recognize x and y as the *eqn* delimiters.

   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 ori **s** to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or

^ to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

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

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

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

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

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

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

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first

column followed by two columns of numerical data. A sample table in this format might be:

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

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

Horizontal lines

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

Vertical lines

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

Space between columns

— A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (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.

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.

Vertical spacing changes

— A key-letter may be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

no effect unless the corresponding table entry is a text block (see below).

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

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

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

Short horizontal lines
— An input table *entry* containing only the string _ is taken to be a single line as

wide as the contents of the column. It is not extended to meet adjoining lines.

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 11 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, spacing and font, using the **p**, **v** and **f** modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings:

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

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 10 and 13. 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 directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *-TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are *-ms* and *-mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

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

Tbl 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	$\operatorname{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}\,dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)\,d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s}$ (Re $s>1$)

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
(Democrats)

Name	Office address	Phone
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(1.5i)p8.
Some Interesting Places

_
Name⓽ Description⓽ Practical Information

_
T{
American Museum of Natural History
T}⓽ T{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors.  There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}⓽ Hours⓽ 10-5, ex. Sun 11-5, Wed. to 9
\^⓽\^⓽ Location⓽ T{
Central Park West & 79th St.
T}
\^⓽\^⓽ Admission⓽ Donation: $1.00 asked
\^⓽\^⓽ Subway⓽ AA to 81st St.
\^⓽\^⓽ Telephone⓽ 212-873-4225

_
Bronx Zoo⓽ T{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}⓽ Hours⓽ T{
10-4:30 winter, to 5:00 summer
T}
\^⓽\^⓽ Location⓽ T{
185th St. & Southern Blvd, the Bronx.
T}
\^⓽\^⓽ Admission⓽ $1.00, but Tu,We,Th free
\^⓽\^⓽ Subway⓽ 2, 5 to East Tremont Ave.
\^⓽\^⓽ Telephone⓽ 212-933-1759

_
Brooklyn Museum⓽ T{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}⓽ Hours⓽ Wed-Sat, 10-5, Sun 12-5
\^⓽\^⓽ Location⓽ T{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^⓽\^⓽ Admission⓽ Free
\^⓽\^⓽ Subway⓽ 2,3 to Eastern Parkway.
\^⓽\^⓽ Telephone⓽ 718-638-5000

_
T{
New-York Historical Society
T}⓽ T{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}⓽ Hours⓽ T{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^⓽\^⓽ Location⓽ T{
Central Park West & 77th St.
T}
\^⓽\^⓽ Admission⓽ Free
\^⓽\^⓽ Subway⓽ AA to 81st St.
\^⓽\^⓽ Telephone⓽ 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. 718-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-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. 54, 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
v V	Vertical spacing change	2
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
_	Short horizontal line	3

A System for Typesetting Mathematics

Brian W. Kernighan and Lorinda L. Cherry

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from i=0 to infinity x sub i = pi over 2

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

"Mathematics is known in the trade as *difficult,* or *penalty, copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cdots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \dfrac{1}{2m\sqrt{ab}} \log \dfrac{\sqrt{a}\,e^{mx} - \sqrt{b}}{\sqrt{a}\,e^{mx} + \sqrt{b}} \\ \dfrac{1}{m\sqrt{ab}} \tanh^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \\ \dfrac{-1}{m\sqrt{ab}} \coth^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \end{cases}$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows

what a user has to type to produce these on our system.)

2. Photocomposition

Photocomposition techniques can be used to solve some of the problems of typesetting mathematics. A phototypesetter is a device which exposes a piece of photographic paper or film, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX operating system[3] works by shining light through a character stencil. The character is made the right size by lenses, and the light beam directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF was designed for setting running text. It also provides all of the facilities that one needs for doing mathematics, such as arbitrary horizontal and vertical motions, line-drawing, size changing, but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an "assembly language," by designing a language for describing mathematical expressions, and compiling it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, "normal" mathematical conventions about operator precedence, parentheses, and the like cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced, for they are not in the half-open interval $(a,b]$. Nor should it assume that that $\sqrt{a+b}$ can be replaced by $(a+b)^{\frac{1}{2}}$, or that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators, and the like. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist: if something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, "standard" things should happen automatically. Someone who types "x=y+z+1" should get "$x=y+z+1$". Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We assume that the typist has a reasonable picture (a two-dimensional representation) of the desired final form, as might be handwritten by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Since our design was not firm, it was also necessary that the program be easy to change at any time.

To make the program easy to build and to change, and to guarantee regularity ("it should work everywhere"), the language is defined by a context-free grammar, described in Section 5. The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days. Construction of a working system sufficient to try significant examples required perhaps a person-month. Since then, we have spent a modest amount of additional time over several years tuning, adding facilities, and occasionally changing the language as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible. TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Thus we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from most details of the particular device and character set currently in use. For example, we let TROFF compute the widths of all strings of characters; we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for

typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called "EQN"), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing x=y+z+1 should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

```
x = y
  + z + 1
```

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde "~" gives a space equal to the normal word spacing in text; a circumflex gives half this much, and a tab charcter spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

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

we write

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

Here spaces are *necessary* in the input to indicate that *sin*, *pi*, *int*, and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks "right"), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically made roman wherever found.

Fractions are specified with the keyword *over:*

```
a+b over c+d+e = 1
```

produces

$$\frac{a+b}{c+d+e}=1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup:*

$$x^2+y^2=z^2$$

is produced by

```
x sup 2 + y sup 2 = z sup 2
```

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: "x sup y sup z" is x^{y^z}. The construct "something *sub* something *sup* something" is recognized as a special case, so "x sub i sup 2" is x_i^2 instead of x_i^2.

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2}=\frac{x^2}{a^2}+\frac{y^2}{b^2}$$

is produced by

```
{partial sup 2 f} over {partial x sup 2} =
x sup 2 over a sup 2 + y sup 2 over b sup 2
```

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the expression. The language defines the precedence of *sup* to be higher than that of *over*, so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqrt* operator for making square roots of the appropriate size: "sqrt a+b" produces $\sqrt{a+b}$, and

```
x = {-b +- sqrt{b sup 2 -4ac}} over 2a
```

is

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqrt* is not useful for tall expressions.

Limits on summations, integrals and similar constructions are specified with the keywords *from* and *to*. To get

$$\sum_{i=0}^{\infty} x_i \to 0$$

we need only type

```
sum from i=0 to inf x sub i -> 0
```

Centering and making the Σ big enough and the limits smaller are all automatic. The *from* and *to* parts are both optional, and the central part (e.g., the Σ) can in fact be anything:

```
lim from {x -> pi /2} ( tan~x) = inf
```

is

$$\lim_{x \to \pi/2} (\tan x) = \infty$$

Again, the braces indicate just what goes into the *from* part.

There is a facility for making braces, brackets, parentheses, and vertical bars of the right height, using the keywords *left* and *right:*

```
left [ x+y over 2a right ]~=~1
```

makes

$$\left[\frac{x+y}{2a} \right] = 1$$

A *left* need not have a corresponding *right,* as we shall see in the next example. Any characters may follow *left* and *right,* but generally only various parentheses and bars are meaningful.

Big brackets, etc., are often used with another facility, called *piles,* which make vertical piles of objects. For example, to get

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

we can type

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

The construction "left {" makes a left brace big enough to enclose the "rpile {...}", which is a right-justified pile of "above ... above ...". "lpile" makes a left-justified pile. There are also centered piles. Because of the recursive language definition, a pile can contain any number of elements; any element of a pile can of course contain piles.

Although EQN makes a valiant attempt to use the right sizes and fonts, there are times when the default assumptions are simply not what is wanted. For instance the italic *sign* in the previous example would conventionally be in roman. Slides and transparencies often require larger characters than normal text. Thus we also provide size and font changing commands: "size 12 bold {A~x~=~y}" will produce $\mathbf{A\ x = y}$. *Size* is followed by a number representing a character size in points. (One point is 1/72 inch; this paper is set in 9 point type.)

If necessary, an input string can be quoted in "...", which turns off grammatical significance, and any font or spacing changes that might otherwise be done on it. Thus we can say

```
lim~ roman "sup" ~x sub n = 0
```

to ensure that the supremum doesn't become a superscript:

$$\lim \sup x_n = 0$$

Diacritical marks, long a problem in traditional typesetting, are straightforward:

$$\dot{x} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \overline{z+Z}$$

is made by typing

```
x dot under + x hat + y tilde
+ X hat + Y dotdot = z+Z bar
```

There are also facilities for globally changing default sizes and fonts, for example for making viewgraphs or for setting chemical equations. The language allows for matrices, and for lining up equations at the same horizontal position.

Finally, there is a definition facility, so a user can say

```
define name "..."
```

at any time in the document; henceforth, any occurrence of the token "name" in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own specifications, for it is quite possible to redefine keywords like *sup* or *over*. Section 6 shows an example of definitions.

The EQN preprocessor reads intermixed text and equations, and passes its output to TROFF. Since TROFF uses lines beginning with a period as control words (e.g., ".ce" means "center the next output line"), EQN uses the sequence ".EQ" to mark the beginning of an equation and ".EN" to mark the end. The ".EQ" and ".EN" are passed through to TROFF untouched, so they can also be used by a knowledgeable user to center equations, number them automatically, etc. By default, however, ".EQ" and ".EN" are simply ignored by TROFF, so by default equations are printed in-line.

".EQ" and ".EN" can be supplemented by TROFF commands as desired; for example, a centered display equation can be produced with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type ".EQ" and ".EN" around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to "#", the input:

```
Let #x sub i#, #y# and #alpha# be positive
```

produces:

Let x_i, y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file "f", one issues the command:

```
eqn f | troff
```

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of "boxes," pieced together in various ways. For example, something with a subscript is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols; lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [ ] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn  : box | eqn box
box  : text
     | { eqn }
     | box OVER box
     | SQRT box
     | box SUB box | box SUP box
     | [ L | C | R ]PILE { list }
     | LEFT text eqn [ RIGHT text ]
     | box [ FROM box ] [ TO box ]
     | SIZE text box
     | [ROMAN | BOLD | ITALIC] box
     | box [HAT | BAR | DOT | DOTDOT | TILDE]
     | DEFINE text text
list : eqn | list ABOVE eqn
text : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn  : box | eqn box
box  : text | { eqn }
```

Anywhere a single character could be used, *any* legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

```
a over b over c ?
```

Is it

```
{a over b} over c
```

or is it

```
a over {b over c} ?
```

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right, because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

```
a sup 2 over b
```

We define *sup* to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse[5].

The output code is generated as the input is scanned. Any time a production of the grammar is recognized, (potentially) some TROFF commands are output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production:

```
text    : TEXT
```

The translation of this is simple. We generate a local name for the string, then hand the name and the string to TROFF, and let TROFF perform the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

```
box    : box OVER box
```

is:

```
Width of output box =
  slightly more than largest input width
Height of output box =
  slightly more than sum of input heights
Base of output box =
  slightly more than height of bottom input box
String describing output box =
  move down;
  move right enough to center bottom box;
  draw bottom box (i.e., copy string for bottom box);
  move up; move left enough to center top box;
  draw top box (i.e., copy string for top box);
  move down and left; draw line full width;
  return to proper base line.
```

Most of the other productions have equally simple semantic actions. Picturing the output as a set of properly placed boxes makes the right sequence of positioning commands quite obvious. The main difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$$\begin{matrix} & & k & j \\ l & & & \\ m & & \mathbf{T} & \\ & & n & i \end{matrix}$$

Grammatically, this is easy: it is sufficient to add a production like

```
box    : TENSOR { list }
```

Semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest—how well EQN sets mathematics, how well it satisfies its goal of being "easy to use," and how easy it was to build.

The first question is easily addressed. This entire paper has been set by the program. Readers can judge for themselves whether it is good enough for their purposes. One of our users commented that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we sometimes leave too much space between a roman letter and an italic one. If we were willing to keep track of the fonts involved, we could do this better more of the time.

Some other weaknesses are inherent in our output device. It is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. One user population consists of mathematicians, chemists, physicists, and computer scientists. Their typical reaction has been something like:

(1) It's easy to write, although I make the following mistakes...

(2) How do I do...?

(3) It botches the following things.... Why don't you fix them?

(4) You really need the following features...

The learning time is short. A few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works.

The second user group is much larger, the secretaries and mathematical typists who were the original target of the system. They tend to be enthusiastic converts. They find the language easy to learn (most are largely self-taught), and have little trouble producing the output they want. They are of course less critical of the esthetics of their output than users trained in mathematics. After a transition period, most find using a computer more interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn't and omit them when they are needed. A common instance is typing

```
f(x sub i)
```

which produces

$$f(x_{i})$$

instead of

$$f(x_{i})$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

```
a sub 0 + b sub 1 over
  {a sub 1 + b sub 2 over
    {a sub 2 + b sub 3 over
      {a sub 3 + ... }}}
```

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - be sup -mx} ~=~
left { lpile {
    1 over {2 mab} ~log~
        {sa emx - sb} over {sa emx + sb}
  above
    1 over mab ~ tanh sup -1 ( sa over sb emx )
  above
    -1 over mab ~ coth sup -1 ( sa over sb emx )
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a high-level language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be implemented in short order, given a compiler-compiler and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

[1] *A Manual of Style,* 12th Edition. University of Chicago Press, 1969. p 295.

[2] *Model C/A/T Phototypesetter.* Graphic Systems, Inc., Hudson, N. H.

[3] Ritchie, D. M., and Thompson, K. L., "The UNIX time-sharing system." *Comm. ACM 17,* 7 (July 1974), 365-375.

[4] Ossanna, J. F., TROFF User's Manual. Bell Laboratories Computing Science Technical Report 54, 1977.

[5] Aho, A. V., and Johnson, S. C., "LR Parsing." *Comp. Surv. 6,* 2 (June 1974), 99-124.

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., 1978.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† operating system.

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.

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on the UNIX operating system. 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

† UNIX is a trademark of AT&T Bell Laboratories.

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
```

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 also a shorthand notation so in-line expressions like π_i^2 can be entered without .EQ and .EN. We will talk about it in section 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 *xsub2* 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:

$$(a^2/b_2)^{\frac{1}{2}}$$

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.

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

‡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*.

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

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

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

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

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

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

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

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

$$x_i \quad x^2$$

$$y_i \quad y^2$$

you have to type

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

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

The 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.

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	∇
...	. . .
,....,	, . . . ,
sum	$\sum$
int	$\int$
prod	$\prod$
union	∪
inter	∩

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* 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
```

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. 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, "NROFF/TROFF User's Manual", Bell Laboratories Computing Science Technical Report #54, 1976.

[2] M. E. Lesk, "Typing Documents on UNIX", Bell Laboratories, 1976.

[3] M. E. Lesk, "TBL — A Program for Setting Tables", Bell Laboratories Computing Science Technical Report #49, 1976.

Writing Tools - The STYLE and DICTION Programs

L. L. Cherry

W. Vesterman

Livingston College
Rutgers University

ABSTRACT

Text processing systems are now in heavy use in many companies to format documents. With many documents stored on line, it has become possible to use computers to study writing style itself and to help writers produce better written and more readable prose. The system of programs described here is an initial step toward such help. It includes programs and a data base designed to produce a stylistic profile of writing at the word and sentence level. The system measures readability, sentence and word length, sentence type, word usage, and sentence openers. It also locates common examples of wordy phrasing and bad diction. The system is useful for evaluating a document's style, locating sentences that may be difficult to read or excessively wordy, and determining a particular writer's style over several documents.

1. Introduction

Computers have become important in the document preparation process, with programs to check for spelling errors and to format documents. As the amount of text stored on line increases, it becomes feasible and attractive to study writing style and to attempt to help the writer in producing readable documents. The system of writing tools described here is a first step toward such help. The system includes programs and a data base to analyze writing style at the word and sentence level. We use the term "style" in this paper to describe the results of a writer's particular choices among individual words and sentence forms. Although many judgements of style are subjective, particularly those of word choice, there are some objective measures that experts agree lead to good style. Three programs have been written to measure some of the objectively definable characteristics of writing style and to identify some commonly misused or unnecessary phrases. Although a document that conforms to the stylistic rules is not guaranteed to be coherent and readable, one that violates all of the rules is likely to be difficult or tedious to read. The program STYLE calculates readability, sentence length variability, sentence type, word usage and sentence openers at a rate of about 400 words per second on a PDP11/70 running the UNIX† Operating System. It assumes that the sentences are well-formed, i. e. that each sentence has a verb and that the subject and verb agree in number. DICTION identifies phrases that are either bad usage or unnecessarily wordy. EXPLAIN acts as a thesaurus for the phrases found by DICTION. Sections 2, 3, and 4 describe the programs; Section 5 gives the results on a cross-section of technical documents; Section 6 discusses accuracy and problems; Section 7 gives implementation details.

† UNIX is a trademark of AT&T Bell Laboratories.

2. STYLE

The program STYLE reads a document and prints a summary of readability indices, sentence length and type, word usage, and sentence openers. It may also be used to locate all sentences in a document longer than a given length, of readability index higher than a given number, those containing a passive verb, or those beginning with an expletive. STYLE is based on the system for finding English word classes or parts of speech, PARTS [1]. PARTS is a set of programs that uses a small dictionary (about 350 words) and suffix rules to partially assign word classes to English text. It then uses experimentally derived rules of word order to assign word classes to all words in the text with an accuracy of about 95%. Because PARTS uses only a small dictionary and general rules, it works on text about any subject, from physics to psychology. Style measures have been built into the output phase of the programs that make up PARTS. Some of the measures are simple counters of the word classes found by PARTS; many are more complicated. For example, the verb count is the total number of verb phrases. This includes phrases like:

has been going
was only going
to go

each of which each counts as one verb. Figure 1 shows the output of STYLE run on a paper by Kernighan and Mashey about the UNIX programming environment [2].

```
programming environment
readability grades:
                        (Kincaid) 12.3  (auto) 12.8  (Coleman-Liau) 11.8  (Flesch) 13.5 (46.3)
sentence info:
                        no. sent 335 no. wds 7419
                        av sent leng 22.1 av word leng 4.91
                        no. questions 0 no. imperatives 0
                        no. nonfunc wds 4362  58.8%   av leng 6.38
                        short sent (<17) 35% (118) long sent (>32)  16% (55)
                        longest sent 82 wds at sent 174; shortest sent 1 wds at sent 117
sentence types:
                        simple  34% (114) complex  32% (108)
                        compound  12% (41) compound-complex  21% (72)
word usage:
                        verb types as % of total verbs
                        tobe  45% (373) aux  16% (133) inf  14% (114)
                        passives as % of non-inf verbs  20% (144)
                        types as % of total
                        prep 10.8% (804) conj 3.5% (262) adv 4.8% (354)
                        noun 26.7% (1983) adj 18.7% (1388) pron 5.3% (393)
                        nominalizations  2 % (155)
sentence beginnings:
                        subject opener: noun (63) pron (43) pos (0) adj (58) art (62) tot  67%
                        prep  12% (39) adv  9% (31)
                        verb  0% (1)  sub_conj  6% (20) conj  1% (5)
                        expletives  4% (13)
```

Figure 1

As the example shows, STYLE output is in five parts. After a brief discussion of sentences, we will describe the parts in order.

2.1. What is a sentence?

Readers of documents have little trouble deciding where the sentences end. People don't even have to stop and think about uses of the character "." in constructions like 1.25, A. J. Jones, Ph.D., i. e., or etc. . When a computer reads a document, finding the end of sentences is not as easy. First we must throw away the printer's marks and formatting commands that litter the text in computer form. Then STYLE defines a sentence as a string of words ending in one of:

. ! ? /.

The end marker "/." may be used to indicate an imperative sentence. Imperative sentences that are not so marked are not identified as imperative. STYLE properly handles numbers with embedded decimal points and commas, strings of letters and numbers with embedded decimal points used for naming computer file names, and the common abbreviations listed in Appendix 1. Numbers that end sentences, like the preceding sentence, cause a sentence break if the next word begins with a capital letter. Initials only cause a sentence break if the next word begins with a capital and is found in the dictionary of function words used by PARTS. So the string

J. D. JONES

does not cause a break, but the string

... system H. The ...

does. With these rules most sentences are broken at the proper place, although occasionally either two sentences are called one or a fragment is called a sentence. More on this later.

2.2. Readability Grades

The first section of STYLE output consists of four readability indices. As Klare points out in [3] readability indices may be used to estimate the reading skills needed by the reader to understand a document. The readability indices reported by STYLE are based on measures of sentence and word lengths. Although the indices may not measure whether the document is coherent and well organized, experience has shown that high indices seem to be indicators of stylistic difficulty. Documents with short sentences and short words have low scores; those with long sentences and many polysyllabic words have high scores. The 4 formulae reported are Kincaid Formula [4], Automated Readability Index [5], Coleman-Liau Formula [6] and a normalized version of Flesch Reading Ease Score [7]. The formulae differ because they were experimentally derived using different texts and subject groups. We will discuss each of the formulae briefly; for a more detailed discussion the reader should see [3].

The Kincaid Formula, given by:

$$Reading\_Grade = 11.8 * syl\_per\_wd + .39 * wds\_per\_sent - 15.59$$

was based on Navy training manuals that ranged in difficulty from 5.5 to 16.3 in reading grade level. The score reported by this formula tends to be in the mid-range of the 4 scores. Because it is based on adult training manuals rather than school book text, this formula is probably the best one to apply to technical documents.

The Automated Readability Index (ARI), based on text from grades 0 to 7, was derived to be easy to automate. The formula is:

$$Reading\_Grade = 4.71 * let\_per\_wd + .5 * wds\_per\_sent - 21.43$$

ARI tends to produce scores that are higher than Kincaid and Coleman-Liau but are usually slightly lower than Flesch.

The Coleman-Liau Formula, based on text ranging in difficulty from .4 to 16.3, is:

$$Reading\_Grade = 5.89 * let\_per\_wd - .3 * sent\_per\_100\_wds - 15.8$$

Of the four formulae this one usually gives the lowest grade when applied to technical documents.

The last formula, the Flesch Reading Ease Score, is based on grade school text covering grades 3 to 12. The formula, given by:

$$Reading\_Score = 206.835 - 84.6 * syl\_per\_wd - 1.015 * wds\_per\_sent$$

is usually reported in the range 0 (very difficult) to 100 (very easy). The score reported by STYLE is scaled to be comparable to the other formulas, except that the maximum grade level reported is set to 17. The Flesch score is usually the highest of the 4 scores on technical documents.

Coke [8] found that the Kincaid Formula is probably the best predictor for technical documents; both ARI and Flesch tend to overestimate the difficulty; Coleman-Liau tend to underestimate. On text in the range of grades 7 to 9 the four formulas tend to be about the same. On easy text the Coleman-Liau formula is probably preferred since it is reasonably accurate at the lower grades and it is safer to present text that is a little too easy than a little too hard.

If a document has particularly difficult technical content, especially if it includes a lot of mathematics, it is probably best to make the text very easy to read, i.e. a lower readability index by shortening the sentences and words. This will allow the reader to concentrate on the technical content and not the long sentences. The user should remember that these indices are estimators; they should not be taken as absolute numbers. STYLE called with "–r number" will print all sentences with an Automated Readability Index equal to or greater than "number".

2.3. Sentence length and structure

The next two sections of STYLE output deal with sentence length and structure. Almost all books on writing style or effective writing emphasize the importance of variety in sentence length and structure for good writing. Ewing's first rule in discussing style in the book *Writing for Results* [9] is:

> "Vary the sentence structure and length of your sentences."

Leggett, Mead and Charvat break this rule into 3 in *Prentice-Hall Handbook for Writers* [10] as follows:

> "34a. Avoid the overuse of short simple sentences."
> "34b. Avoid the overuse of long compound sentences."
> "34c. Use various sentence structures to avoid monotony and increase effectiveness."

Although experts agree that these rules are important, not all writers follow them. Sample technical documents have been found with almost no sentence length or type variability. One document had 90% of its sentences about the same length as the average; another was made up almost entirely of simple sentences (80%).

The output sections labeled "sentence info" and "sentence types" give both length and structure measures. STYLE reports on the number and average length of both sentences and words, and number of questions and imperative sentences (those ending in "/."). The measures of non-function words are an attempt to look at the content words in the document. In English non-function words are nouns, adjectives, adverbs, and non-auxiliary verbs; function words are prepositions, conjunctions, articles, and auxiliary verbs. Since most function words are short, they tend to lower the average word length. The average length of non-function words may be a more useful measure for comparing word choice of different writers than the total average word length. The percentages of short and long sentences measure sentence length variability. Short sentences are those at least 5 words less than the average; long sentences are those at least 10 words longer than the average. Last in the sentence information section is the length and location of the longest and shortest sentences. If the flag "–l number" is used, STYLE will print all sentences longer than "number".

Because of the difficulties in dealing with the many uses of commas and conjunctions in English, sentence type definitions vary slightly from those of standard textbooks, but still measure the same constructional activity.

1. A simple sentence has one verb and no dependent clause.
2. A complex sentence has one independent clause and one dependent clause, each with one verb. Complex sentences are found by identifying sentences that contain either a subordinate conjunction or a clause beginning with words like "that" or "who". The preceding sentence has such a clause.
3. A compound sentence has more than one verb and no dependent clause. Sentences joined by ";" are also counted as compound.
4. A compound-complex sentence has either several dependent clauses or one dependent clause and a compound verb in either the dependent or independent clause.

Even using these broader definitions, simple sentences dominate many of the technical documents that have been tested, but the example in Figure 1 shows variety in both sentence structure and sentence length.

2.4. Word Usage

The word usage measures are an attempt to identify some other constructional features of writing style. There are many different ways in English to say the same thing. The constructions differ from one another in the form of the words used. The following sentences all convey approximately the same meaning but differ in word usage:

> The cxio program is used to perform all communication between the systems.
> The cxio program performs all communications between the systems.
> The cxio program is used to communicate between the systems.
> The cxio program communicates between the systems.
> All communication between the systems is performed by the cxio program.

The distribution of the parts of speech and verb constructions helps identify overuse of particular constructions. Although the measures used by STYLE are crude, they do point out problem areas. For each category, STYLE reports a percentage and a raw count. In addition to looking at the percentage, the user may find it useful to compare the raw count with the number of sentences. If, for example, the number of infinitives is almost equal to the number of sentences, then many of the sentences in the document are constructed like the first and third in the preceding example. The user may want to transform some of these sentences into another form. Some of the implications of the word usage measures are discussed below.

Verbs are measured in several different ways to try to determine what types of verb constructions are most frequent in the document. Technical writing tends to contain many passive verb constructions and other usage of the verb "to be". The category of verbs labeled "tobe" measures both passives and sentences of the form:

subject tobe predicate

In counting verbs, whole verb phrases are counted as one verb. Verb phrases containing auxiliary verbs are counted in the category "aux". The verb phrases counted here are those whose tense is not simple present or simple past. It might eventually be useful to do more detailed measures of verb tense or mood. Infinitives are listed as "inf". The percentages reported for these three categories are based on the total number of verb phrases found. These categories are not mutually exclusive; they cannot be added, since, for example, "to be going" counts as both "tobe" and "inf". Use of these three types of verb constructions varies significantly among authors.

STYLE reports passive verbs as a percentage of the finite verbs in the document. Most style books warn against the overuse of passive verbs. Coleman [11] has shown that sentences with active verbs are easier to learn than those with passive verbs. Although the inverted object-subject order of the passive voice seems to emphasize the object, Coleman's experiments showed that there is little difference in retention by word position. He also

showed that the direct object of an active verb is retained better than the subject of a passive verb. These experiments support the advice of the style books suggesting that writers should try to use active verbs wherever possible. The flag "–p" causes STYLE to print all sentences containing passive verbs.

Pronouns

add cohesiveness and connectivity to a document by providing back-reference. They are often a short-hand notation for something previously mentioned, and therefore connect the sentence containing the pronoun with the word to which the pronoun refers. Although there are other mechanisms for such connections, documents with no pronouns tend to be wordy and to have little connectivity.

Adverbs

can provide transition between sentences and order in time and space. In performing these functions, adverbs, like pronouns, provide connectivity and cohesiveness.

Conjunctions

provide parallelism in a document by connecting two or more equal units. These units may be whole sentences, verb phrases, nouns, adjectives, or prepositional phrases. The compound and compound-complex sentences reported under sentence type are parallel structures. Other uses of parallel structures are indicated by the degree that the number of conjunctions reported under word usage exceeds the compound sentence measures.

Nouns and Adjectives.

A ratio of nouns to adjectives near unity may indicate the over-use of modifiers. Some technical writers qualify every noun with one or more adjectives. Qualifiers in phrases like "simple linear single-link network model" often lend more obscurity than precision to a text.

Nominalizations

are verbs that are changed to nouns by adding one of the suffixes "ment", "ance", "ence", or "ion". Examples are accomplishment, admittance, adherence, and abbreviation. When a writer transforms a nominalized sentence to a non-nominalized sentence, she/he increases the effectiveness of the sentence in several ways. The noun becomes an active verb and frequently one complicated clause becomes two shorter clauses. For example,

> Their inclusion of this provision is admission of the importance of the system.
> When they included this provision, they admitted the importance of the system.

Coleman found that the transformed sentences were easier to learn, even when the transformation produced sentences that were slightly longer, provided the transformation broke one clause into two. Writers who find their document contains many nominalizations may want to transform some of the sentences to use active verbs.

2.5. Sentence openers

Another agreed upon principle of style is variety in sentence openers. Because STYLE determines the type of sentence opener by looking at the part of speech of the first word in the sentence, the sentences counted under the heading "subject opener" may not all really begin with the subject. However, a large percentage of sentences in this category still indicates lack of variety in sentence openers. Other sentence opener measures help the user determine if there are transitions between sentences and where the subordination occurs. Adverbs and conjunctions at the beginning of sentences are mechanisms for transition between sentences. A pronoun at the beginning shows a link to something previously mentioned and indicates connectivity.

The location of subordination can be determined by comparing the number of sentences that begin with a subordinator with the number of sentences with complex clauses. If few sentences start with subordinate conjunctions then the subordination is embedded or at the end of the complex sentences. For variety the writer may want to transform some sentences to have leading subordination.

The last category of openers, expletives, is commonly overworked in technical writing. Expletives are the words "it" and "there", usually with the verb "to be", in constructions where the subject follows the verb. For example,

> There are three streets used by the traffic.
> There are too many users on this system.

This construction tends to emphasize the object rather than the subject of the sentence. The flag "–e" will cause STYLE to print all sentences that begin with an expletive.

3. DICTION

The program DICTION prints all sentences in a document containing phrases that are either frequently misused or indicate wordiness. The program, an extension of Aho's FGREP [12] string matching program, takes as input a file of phrases or patterns to be matched and a file of text to be searched. A data base of about 450 phrases has been compiled as a default pattern file for DICTION. Before attempting to locate phrases, the program maps upper case letters to lower case and substitutes blanks for punctuation. Sentence boundaries were deemed less critical in DICTION than in STYLE, so abbreviations and other uses of the character "." are not treated specially. DICTION brackets all pattern matches in a sentence with the characters "[" "]" . Although many of the phrases in the default data base are correct in some contexts, in others they indicate wordiness. Some examples of the phrases and suggested alternatives are:

Phrase	Alternative
a large number of	many
arrive at a decision	decide
collect together	collect
for this reason	so
pertaining to	about
through the use of	by or with
utilize	use
with the exception of	except

Appendix 2 contains a complete list of the default file. Some of the entries are short forms of problem phrases. For example, the phrase "the fact" is found in all of the following and is sufficient to point out the wordiness to the user:

Phrase	Alternative
accounted for by the fact that	caused by
an example of this is the fact that	thus
based on the fact that	because
despite the fact that	although
due to the fact that	because
in light of the fact that	because
in view of the fact that	since
notwithstanding the fact that	although

Entries in Appendix 2 preceded by "~" are not matched. See Section 7 for details on the use of "~".

The user may supply her/his own pattern file with the flag "–f patfile". In this case the default file will be loaded first, followed by the user file. This mechanism allows users to suppress patterns contained in the default file or to include their own pet peeves that are not in the default file. The flag "–n" will exclude the default file altogether. In constructing a pattern file, blanks should be used before and after each phrase to avoid matching substrings in words. For example, to find all occurrences of the word "the", the pattern " the " should be used. The blanks cause only the word "the" to be matched and not the string "the" in words like there, other, and

therefore. One side effect of surrounding the words with blanks is that when two phrases occur without intervening words, only the first will be matched.

4. EXPLAIN

The last program, EXPLAIN, is an interactive thesaurus for phrases found by DICTION. The user types one of the phrases bracketed by DICTION and EXPLAIN responds with suggested substitutions for the phrase that will improve the diction of the document.

Table 1
Text Statistics on 20 Technical Documents

	variable	minimum	maximum	mean	standard deviation
Readability	Kincaid	9.5	16.9	13.3	2.2
	automated	9.0	17.4	13.3	2.5
	Cole-Liau	10.0	16.0	12.7	1.8
	Flesch	8.9	17.0	14.4	2.2
sentence info.	av sent length	15.5	30.3	21.6	4.0
	av word length	4.61	5.63	5.08	.29
	av nonfunction length	5.72	7.30	6.52	.45
	short sent	23%	46%	33%	5.9
	long sent	7%	20%	14%	2.9
sentence types	simple	31%	71%	49%	11.4
	complex	19%	50%	33%	8.3
	compound	2%	14%	7%	3.3
	compound-complex	2%	19%	10%	4.8
verb types	tobe	26%	64%	44.7%	10.3
	auxiliary	10%	40%	21%	8.7
	infinitives	8%	24%	15.1%	4.8
	passives	12%	50%	29%	9.3
word usage	prepositions	10.1%	15.0%	12.3%	1.6
	conjunction	1.8%	4.8%	3.4%	.9
	adverbs	1.2%	5.0%	3.4%	1.0
	nouns	23.6%	31.6%	27.8%	1.7
	adjectives	15.4%	27.1%	21.1%	3.4
	pronouns	1.2%	8.4%	2.5%	1.1
	nominalizations	2%	5%	3.3%	.8
sentence openers	prepositions	6%	19%	12%	3.4
	adverbs	0%	20%	9%	4.6
	subject	56%	85%	70%	8.0
	verbs	0%	4%	1%	1.0
	subordinating conj	1%	12%	5%	2.7
	conjunctions	0%	4%	0%	1.5
	expletives	0%	6%	2%	1.7

5. Results

5.1. STYLE

To get baseline statistics and check the program's accuracy, we ran STYLE on 20 technical documents. There were a total of 3287 sentences in the sample. The shortest document was 67 sentences long; the longest 339 sentences. The documents covered a wide range of subject matter, including theoretical computing, physics, psychology, engineering, and affirmative action. Table 1 gives the range, median, and standard deviation of the various style measures. As you will note most of the measurements have a fairly wide range of values across the sample documents.

As a comparison, Table 2 gives the median results for two different technical authors, a sample of instructional material, and a sample of the Federalist Papers. The two authors show similar styles, although author 2 uses somewhat shorter sentences and longer words than author 1. Author 1 uses all types of sentences, while author 2 prefers simple and complex sentences, using few compound or compound-complex sentences. The other major difference in the styles of these authors is the location of subordination. Author 1 seems to prefer embedded or trailing subordination, while author 2 begins many sentences with the subordinate clause. The documents tested for both authors 1 and 2 were technical documents, written for a technical audience. The instructional documents, which are written for craftspeople, vary surprisingly little from the two technical samples. The sentences and words are a little longer, and they contain many passive and auxiliary verbs, few adverbs, and almost no pronouns. The instructional documents contain many imperative sentences, so there are many sentence with verb openers. The sample of Federalist Papers contrasts with the other samples in almost every way.

Table 2
Text Statistics on Single Authors

	variable	author 1	author 2	inst.	FED
readability	Kincaid	11.0	10.3	10.8	16.3
	automated	11.0	10.3	11.9	17.8
	Coleman-Liau	9.3	10.1	10.2	12.3
	Flesch	10.3	10.7	10.1	15.0
sentence info	av sent length	22.64	19.61	22.78	31.85
	av word length	4.47	4.66	4.65	4.95
	av nonfunction length	5.64	5.92	6.04	6.87
	short sent	35%	43%	35%	40%
	long sent	18%	15%	16%	21%
sentence types	simple	36%	43%	40%	31%
	complex	34%	41%	37%	34%
	compound	13%	7%	4%	10%
	compound-complex	16%	8%	14%	25%
verb type	tobe	42%	43%	45%	37%
	auxiliary	17%	19%	32%	32%
	infinitives	17%	15%	12%	21%
	passives	20%	19%	36%	20%
word usage	prepositions	10.0%	10.8%	12.3%	15.9%
	conjunctions	3.2%	2.4%	3.9%	3.4%
	adverbs	5.05%	4.6%	3.5%	3.7%
	nouns	27.7%	26.5%	29.1%	24.9%
	adjectives	17.0%	19.0%	15.4%	12.4%
	pronouns	5.3%	4.3%	2.1%	6.5%
	nominalizations	1%	2%	2%	3%
sentence openers	prepositions	11%	14%	6%	5%
	adverbs	9%	9%	6%	4%
	subject	65%	59%	54%	66%
	verb	3%	2%	14%	2%
	subordinating conj	8%	14%	11%	3%
	conjunction	1%	0%	0%	3%
	expletives	3%	3%	0%	3%

5.2. DICTION

In the few weeks that DICTION has been available to users about 35,000 sentences have been run with about 5,000 string matches. The authors using the program seem to make the suggested changes about 50-75% of the time. To date, almost 200 of the 450 strings in the default file have been matched. Although most of these phrases are valid and correct in some contexts, the 50-75% change rate seems to show that the phrases are used much more often than concise diction warrants.

6. Accuracy

6.1. Sentence Identification

The correctness of the STYLE output on the 20 document sample was checked in detail. STYLE misidentified 129 sentence fragments as sentences and incorrectly joined two or more sentences 75 times in the 3287 sentence sample. The problems were usually because of nonstandard formatting commands, unknown abbreviations, or lists of non-sentences. An impossibly long sentence found as the longest sentence in the document usually is the result of a long list of non-sentences.

6.2. Sentence Types

Style correctly identified sentence type on 86.5% of the sentences in the sample. The type distribution of the sentences was 52.5% simple, 29.9% complex, 8.5% compound and 9% compound-complex. The program reported 49.5% simple, 31.9% complex, 8% compound and 10.4% compound-complex. Looking at the errors on the individual documents, the number of simple sentences was under-reported by about 4% and the complex and compound-complex were over-reported by 3% and 2%, respectively. The following matrix shows the programs output vs. the actual sentence type.

		Program Results			
		simple	complex	compound	comp-complex
Actual	simple	1566	132	49	17
Sentence	complex	47	892	6	65
Type	compound	40	6	207	23
	comp-complex	0	52	5	249

The system's inability to find imperative sentences seems to have little effect on most of the style statistics. A document with half of its sentences imperative was run, with and without the imperative end marker. The results were identical except for the expected errors of not finding verbs as sentence openers, not counting the imperative sentences, and a slight difference (1%) in the number of nouns and adjectives reported.

6.3. Word Usage

The accuracy of identifying word types reflects that of PARTS, which is about 95% correct. The largest source of confusion is between nouns and adjectives. The verb counts were checked on about 20 sentences from each document and found to be about 98% correct.

7. Technical Details

7.1. Finding Sentences

The formatting commands embedded in the text increase the difficulty of finding sentences. Not all text in a document is in sentence form; there are headings, tables, equations and lists, for example. Headings like "Finding Sentences" above should be discarded, not attached to the next sentence. However, since many of the documents are formatted to be phototypeset, and contain font changes, which usually operate on the most important words in the document, discarding all

formatting commands is not correct. To improve the programs' ability to find sentence boundaries, the deformatting program, DEROFF [13], has been given some knowledge of the formatting packages used on the UNIX operating system. DEROFF will now do the following:

1. Suppress all formatting macros that are used for titles, headings, author's name, etc.
2. Suppress the arguments to the macros for titles, headings, author's name, etc.
3. Suppress displays, tables, footnotes and text that is centered or in no-fill mode.
4. Substitute a place holder for equations and check for hidden end markers. The place holder is necessary because many typists and authors use the equation setter to change fonts on important words. For this reason, header files containing the definition of the EQN delimiters must also be included as input to STYLE. End markers are often hidden when an equation ends a sentence and the period is typed inside the EQN delimiters.
5. Add a "." after lists. If the flag –ml is also used, all lists are suppressed. This is a separate flag because of the variety of ways the list macros are used. Often, lists are sentences that should be included in the analysis. The user must determine how lists are used in the document to be analyzed.

Both STYLE and DICTION call DEROFF before they look at the text. The user should supply the –ml flag if the document contains many lists of non-sentences that should be skipped.

7.2. Details of DICTION

The program DICTION is based on the string matching program FGREP. FGREP takes as input a file of patterns to be matched and a file to be searched and outputs each line that contains any of the patterns with no indication of which pattern was matched. The following changes have been added to FGREP:

1. The basic unit that DICTION operates on is a sentence rather than a line. Each sentence that contains one of the patterns is output.
2. Upper case letters are mapped to lower case.
3. Punctuation is replaced by blanks.
4 All pattern matches in the sentence are found and surrounded with "[" "]" .
5. A method for suppressing a string match has been added. Any pattern that begins with "~" will not be matched. Because the matching algorithm finds the longest substring, the suppression of a match allows words in some correct contexts not to be matched while allowing the word in another context to be found. For example, the word "which" is often incorrectly used instead of "that" in restrictive clauses. However, "which" is usually correct when preceded by a preposition or ",". The default pattern file suppresses the match of the common prepositions or a double blank followed by "which" and therefore matches only the suspect uses. The double blank accounts for the replaced comma.

8. Conclusions

A system of writing tools that measure some of the objective characteristics of writing style has been developed. The tools are sufficiently general that they may be applied to documents on any subject with equal accuracy. Although the measurements are only of the surface structure of the text, they do point out problem areas. In addition to helping writers produce better documents, these programs may be useful for studying the writing process and finding other formulae for measuring readability.

References

1. L. L. Cherry, "PARTS - A System for Assigning Word Classes to English Text," submitted *Communications of the ACM.*
2. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," *Software - Practice & Experience* , **9**, 1-15 (1979).
3. G. R. Klare, "Assessing Readability," *Reading Research Quarterly,* 1974-1975, **10** , 62-102.
4. E. A. Smith and P. Kincaid, "Derivation and validation of the automated readability index for use with technical materials," *Human Factors,* 1970, 12, 457-464.
5. J. P. Kincaid, R. P. Fishburne, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (Automated Readability Index, Fog count, and Flesch Reading Ease Formula) for Navy enlisted personnel," Navy Training Command Research Branch Report 8-75, Feb., 1975.
6. M. Coleman and T. L. Liau, "A Computer Readability Formula Designed for Machine Scoring," *Journal of Applied Psychology,* 1975, 60, 283-284.
7. R. Flesch, "A New Readability Yardstick," *Journal of Applied Psychology,* 1948, 32, 221-233.
8. E. U. Coke, private communication.
9. D. W. Ewing, *Writing for Results,* John Wiley & Sons, Inc., New York, N. Y. (1974).
10. G. Leggett, C. D. Mead and W. Charvat, *Prentice-Hall Handbook for Writers,* Seventh Edition, Prentice-Hall Inc., Englewood Cliffs, N. J. (1978).
11. E. B. Coleman, "Learning of Prose Written in Four Grammatical Transformations," *Journal of Applied Psychology,* 1965, vol. 49, no. 5, pp. 332-341.

12 A. V. Aho and M. J. Corasick, "Efficient String Matching: an aid to Bibliographic Search," *Communications of the ACM,* **18**, (6), 333-340, June 1975.

13. Bell Laboratories, *"UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL,"* Seventh Edition, Vol. 1 (January 1979).

Appendix 1

STYLE Abbreviations

a. d.
A. M.
a. m.
b. c.
Ch.
ch.
ckts.
dB.
Dept.
dept.
Depts.
depts.
Dr.
Drs.
e. g.
Eq.
eq.
et al.
etc.
Fig.
fig.
Figs.
figs.
ft.
i. e.
in.
Inc.
Jr.
jr.
mi.
Mr.
Mrs.
Ms.
No.
no.
Nos.
nos.
P. M.
p. m.
Ph. D.
Ph. d.
Ref.
ref.
Refs.
refs.
St.
vs.
yr.

Appendix 2

Default DICTION Patterns

a great deal of
a large number of
a lot of
a majority of
a need for
a number of
a particular preference for
a preference for
a small number of
a tendency to
abovementioned
absolutely complete
absolutely essential
accomplished
accordingly
activate
actual
added increments
adequate enough
advent
afford an opportunity
aggregate
all of
all throughout
along the line
an indication of
analyzation
and etc
and or
another additional
any and all
arrive at a
as a matter of fact
as a method of
as good or better than
as of now
as per
as regards
as related to
as to
assistance
assistance to
assistance to
assuming that
at a later date
at about
at above
at all times
at an early date
at below
at the present
at the time when
at this point in time
at this time
at which time
at your earliest convenience
authorization
awful
basic fundamentals
basically
be cognizant of
being as
being that
brief in duration
bring to a conclusion
but that
but what
by means of
by the use of
carry out experiments
center about
center around
center portion
check into
check on
check up on
circle around
close proximity
collaborate together
collect together
combine together
come to an end
commence
common accord
compensation
completely eliminated
comprise
concerning
conduct an investigation of
conjecture
connect up
consensus of opinion
consequent result
consolidate together
construct
contemplate
continue on
continue to remain
could of
count up
couple together
debate about
decide on
deleterious effect
demean
demonstrate
depreciate in value
deserving of
desirable benefits
desirous of
different than
discontinue
disutility
divide up
doubt but
due to
duly noted
during the time that
each and every
early beginnings
effectuate
emotional feelings
empty out
enclosed herein
enclosed herewith
end result
end up
endeavor
enter in
enter into
enthused
entirely complete
equally good as
essentially
eventuate
every now and then
exactly identical
experiencing difficulty
fabricate
face up to
facilitate
facts and figures
fast in action
fearful of
fearful that
few in number
file away
final completion
final ending
final outcome
final result
finalize
find it interesting to know
first and foremost
first beginnings
first initiated
firstly
follow after
following after
for the purpose of
for the reason that
for the simple reason that
for this reason
for your information
from the point of view of
full and complete
generally agreed
good and
got to
gratuitous
greatly minimize
head up
help but
helps in the production of
hopeful
if and when
if at all possible
impact
implement
important essentials
importantly
in a large measure
in a position to
in accordance
in advance of
in agreement with
in all cases
in back of
in behalf of
in behind
in between
in case
in close proximity
in conflict with
in conjunction with
in connection with
in fact
in large measure
in many cases
in most cases
in my opinion I think
in order to
in rare cases
in reference to
in regard to
in regards to
in relation with
in short supply
in size
in terms of
in the amount of
in the case of
in the course of
in the event
in the field of
in the form of
in the instance of
in the interim
in the last analysis
in the matter of
in the near future
in the neighborhood of
in the not too distant future
in the proximity of
in the range of
in the same way as described
in the shape of
in the vicinity of
in this case
in view of the
in violation of
inasmuch as
indicate
indicative of
initialize
initiate
injurious to
inquire
inside of
institute a
intents and purposes
intermingle
irregardless
is defined as
is used to control
is when
is where
it is incumbent
it stands to reason
it was noted that if
joint cooperation
joint partnership
just exactly
kind of
know about
last but not least
later on
leaving out of consideration
liable
link up
literally
little doubt that
lose out on
lots of
main essentials
make a
make adjustments to
make an
make application to
make contact with
make mention of
make out a list of
make the acquaintance of
make the adjustment
manner
maximum possible
meaningful
meet up with
melt down
melt up
methodology
might of
minimize as far as possible
minor importance
miss out on
modification

more preferable
most unique
must of
mutual cooperation
necessary requisite
necessitate
need for
nice
not be un
not in a position to
not of a high order of accuracy
not un
notwithstanding
of considerable magnitude
of that
of the opinion that
off of
on a few occasions
on account of
on behalf of
on the grounds that
on the occasion
on the part of
one of the
open up
operates to correct
outside of
over with
overall
past history
perceptive of
perform a measurement
perform the measurement
permits the reduction of
personalize
pertaining to
physical size
plan ahead
plan for the future
plan in advance
plan on
present a conclusion
present a report
presently
prior to
prioritize
proceed to
procure
productive of
prolong the duration
protrude out from
provided that
pursuant to
put to use in
range all the way from
reason is because
reason why
recur again
reduce down
refer back
reference to this
reflective of
regarding
regretful
reinitiate
relative to
repeat again
representative of
resultant effect
resume again
retreat back
return again
return back
revert back
seal off
seems apparent
send a communication
short space of time
should of
single unit
situation
so as to
sort of
spell out
still continue
still remain
subsequent
substantially in agreement
succeed in
suggestive of
superior than
surrounding circumstances
take appropriate
take cognizance of
take into consideration
termed as
terminate
termination
the author
the authors
the case that
the fact
the foregoing
the foreseeable future
the fullest possible extent
the majority of
the nature
the necessity of
the only difference being that
the order of
the point that
the truth is
there are not many
through the medium of
through the use of
throughout the entire
time interval
to summarize the above
total effect of all this
totality
transpire
true facts
try and
ultimate end
under a separate cover
under date of
under separate cover
under the necessity to
underlying purpose
undertake a study
uniformly consistent
unique
until such time as
up to this time
upshot
utilize
very
very complete
very unique
vital
whlch
with a view to
with reference to
with regard to
with the exception of
with the object of
with the result that
with this in mind, it is clear that
within the realm of possibility
without further delay
worth while
would of
ing behavior
wise
~ which
~ about which
~ after which
~ at which
~ between which
~ by which
~ for which
~ from which
~ in which
~ into which
~ of which
~ on which
~ on which
~ over which
~ through which
~ to which
~ under which
~ upon which
~ with which
~ without which
~clockwise
~likewise
~otherwise

Part 4
Supporting Tools

BC – An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

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.

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 [5]) 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]. Those who are familiar with C will find few surprises in this language.

† UNIX is a trademark of AT&T Bell Laboratories.

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] 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 [3].

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 [4]; its original version was written by S. C. Johnson.

References

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines,* Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978.

[5] 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:

$$\min(a \times b, \max(\mathbf{scale}, a))$$

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:

$$\min(a+b, \max(\mathbf{scale}, a, b))$$

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

ABSTRACT

DC is an interactive desk calculator program implemented on the UNIX† time-sharing system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

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.

† UNIX is a trademark of AT&T Bell Laboratories.

+ – * % ^

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

$$x_{n+1} = \tfrac{1}{2}\left(x_n + \frac{y}{x_n}\right)$$

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 are 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 _ (an underscore). 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 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).

An Introduction to the Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions.

1. RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original -- previous revisions remain accessible.
   a. Maintains each module as a tree of revisions.
   b. Project libraries can be organized centrally, decentralized, or any way you like.
   c. RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
2. RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.
   a. RCS performs automatic record keeping.
   b. RCS logs all changes automatically.
   c. RCS guarantees project continuity.
3. RCS manages multiple lines of development.
4. RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.
5. RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.
6. RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other one.
7. RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
8. RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.

9. RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a given configuration.
10. Provides high-level management visibility. Thus, it is easy to track the status of a software project.
    a. RCS provides a complete change history.
    b. RCS records who did what when to which revision of which module.
11. RCS is fully compatible with existing software development tools. RCS is unobtrusive -- its interface to the file system is such that all your existing software tools can be used as before.
12. RCS' basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.
13. RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
14. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke the checkin command:

```
ci f.c
```

This command creates f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ,v are called RCS files ("v" stands for "versions"), the others are called working files. To get back the working file f.c in the previous example, use the checkout command:

```
co f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it in back in by invoking:

```
ci f.c
```

Ci increments the revision number properly. If *ci* complains with the message

```
ci error: no lock set by <your login>
```

then your system administrator has decided to create all RCS files with the locking attribute set to "strict". With strict locking, you you must lock the revision during the previous checkout. Thus, your last checkout should have been

```
co -l f.c
```

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout would overwrite your changes. Instead, invoke

```
rcs -l f.c
```

This command will lock the latest revision for you, unless somebody else got ahead of you

already. If someone else has the lock you will have to negotiate your changes with them.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner off the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the commands:

rcs -U f.c and rcs -L f.c

You can set the locking to strict or non-strict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any change*.

To avoid the deletion of the working file during checkin (should you want to continue editing), invoke

ci -l f.c

This command checks in f.c as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. There is also an option -u for *ci* that does a checkin followed by a checkout without locking. This is useful if you want to compile the file after the checkin. Both options also update the identification markers in your file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

ci -r2 f.c or ci -r2.1 f.c

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

co -r2 f.c and co -r2.1 f.c

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

ci -r1.3.1 f.c

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see *rcsfile*(5).

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

$Header$

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

$Header: filename revisionnumber date time author state $

You never need to touch this string, because RCS keeps it up to date automatically. To propagate the marker into your object code, simply put it into a literal character string. In C, this is done as follows:

* Pairs of RCS and working files can really be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. Both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

```
static char rcsid[] = "$Header$";
```

The command *ident* extracts such markers from any file, even object code. Thus, *ident* helps you to find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker

```
$Log$
```

into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see *co* (1) for details.

How to combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form .c,v.c, because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use this:

```
.SUFFIXES: .c,v

.c,v.o:
        co -q $*.c
        cc $(CFLAGS) -c $*.c
        rm -f $*.c

prog:   f1.o f2.o .....
        cc f1.o f2.o ..... -o prog
```

This rule has the following effect. If a file f.c does not exist, and f.o is older than f.c,v, MAKE checks out f.c, compiles f.c into f.o, and then deletes f.c. From then on, MAKE will use f.o until you change f.c,v.

If f.c exists (presumably because you are working on it), the default rule .c.o takes precedence, and f.c is compiled into f.o, but not deleted.

If you keep your RCS file in the directory ./RCS, all this will not work and you have to write explicit checkout rules for every file, like

```
f1.c:   RCS/f1.c,v; co -q f1.c
```

Unfortunately, these rules do not have the property of removing unneeded .c-files.

Additional Information on RCS

If you want to know more about RCS, for example how to work with a tree of revisions and how to use symbolic revision numbers, read the following paper:

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Taking a look at the manual page *RCSFILE*(5) should also help to understand the revision tree permitted by RCS.

Lex – A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

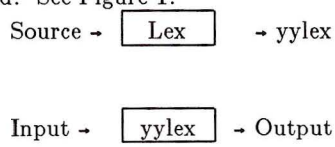
1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. 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.



An overview of Lex
Figure 1

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

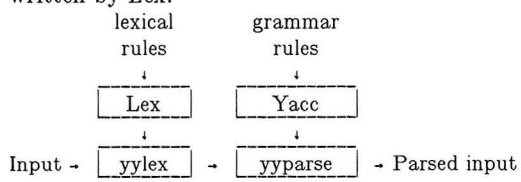
```
%%
[ \t]+$    ;
```

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.



Lex with Yacc
Figure 2

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 taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the ***actions*** to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ***ab*** and another for ***abcdefg***, and the input stream is ***abcdefh***, Lex will recognize ***ab*** and leave the input pointer just before ***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. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ^. The – character indicates ranges. For example,

[a–z0–9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, 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 nonportable:

[\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 *abcfef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string *ab*, but only if followed by *cd*. Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of *a*.

Finally, initial % is special, being the separator for Lex source segments.

4. Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

[\t\n] ;

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a-z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+    printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output.

This action is so common that it may be written as ECHO:

```
[a–z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches ***read*** it will normally match the instances of ***read*** contained in ***bread*** or ***readjust***; to avoid this, a rule of the form */a–z/+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count ***yyleng*** of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a–zA–Z]+    {words++; chars += yyleng;}
```

which accumulates in ***chars*** the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng–1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, ***yymore()*** can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in ***yytext***. Second, ***yyless (n)*** may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument ***n*** indicates the number of characters in ***yytext*** to be retained. Further characters previously matched are returned to the input. This provides the same sort of look-ahead 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("Op (=–) 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("Op (=–) ambiguous\n");
                yyless(yyleng–2);
                ... action for = ...
                }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=–/[A–Za–z]
```

in the first case and

```
=/–[A–Za–z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=–3", however, makes

```
=–/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) ***input()*** which returns the next input character;
2) ***output(c)*** which writes the character ***c*** on the output; and

3) ***unput(c)*** pushes the character ***c*** back onto the input stream to be read later by ***input()***.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by ***input*** must mean end of file; and the relationship between ***unput*** and ***input*** must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ 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.

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                       [DEde][-+]?{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*. The I/O library is defined in terms of the C standard library [6].

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.

UNIX. The library is accessed by the loader flag *-ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

8. Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a 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
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
        int k;
```

```
[0–9]+    {
          k = atoi(yytext);
          if (k%7 == 0)
              printf("%d", k+3);
          else
              printf("%d",k);
          }
```

to do just that. The rule [0–9]+ recognizes strings of digits; ***atoi*** converts the digits to binary and stores the result in ***k***. The operator % (remainder) is used to check whether ***k*** is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as ***49.63*** or ***X7***. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                    int k;
-?[0–9]+            {
                    k = atoi(yytext);
                    printf("%d",
                      k%7 == 0 ? k+3 : k);
                    }
-?[0–9.]+           ECHO;
[A-Za-z][A-Za-z0-9]+ ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The ***if–else*** has been replaced by a C conditional expression to save space; the form ***a?b:c*** means "if ***a*** then ***b*** else ***c***".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
int lengs[100];
%%
[a–z]+    lengs[yyleng]++;
.         |
\n        ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement ***return(1);*** indicates that Lex is to perform wrapup. If ***yywrap*** returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a ***yywrap*** that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a    [aA]
b    [bB]
c    [cC]
...
z    [zZ]
```

An additional class recognizes white space:

```
W    [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0–9]+{W}{d}{W}[+–]?{W}[0–9]+     |
[0–9]+{W}"."{W}{d}{W}[+–]?{W}[0–9]+     |
"."{W}[0–9]+{W}{d}{W}[+–]?{W}[0–9]+     {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p=+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the ***for*** loop to find

the letter *d* or *D*. The program 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}1{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. Normally, the letter ***a*** is represented as the same form as the character constant ***'a'***. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Sample character table.

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 rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form

```
%{
```

code
%}

4) Start conditions, given in the form
%S name1 name2 ...

5) Character set tables, in the form
%T
number space character-string
...
%T

6) Changes to internal array sizes, in the form
%x nnn
where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used ***unput*** to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

14. Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15. References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran,* Software – Practice and Experience, 5, pp. 395-496 (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,* Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

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.

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

† UNIX is a trademark of AT&T Bell Laboratories.

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 **n**th 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.

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially 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

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

Revised April, 1986

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 *lS* 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 -lS"
```

loads them with both the Lex ("-ll") and the Standard ("-lS") 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.

† UNIX is a trademark of AT&T Bell Laboratories.

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 is 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 -lS
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., *cd* 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).

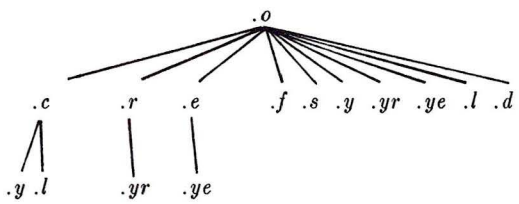
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

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.

Another special macro is 'VPATH'. The "VPATH" macro should be set to a list of directories separated by colons. When *make* searches for a file as a result of a dependency relation, it will first search the current directory and then each of the directories on the "VPATH" list. If the file is found, the actual path to the file will be used, rather than just the filename. If "VPATH" is not defined, then only the current directory is searched. Note that "VPATH" is intended to act like the System V "VPATH" support, but there is no guarantee that it functions identically.

One use for "VPATH" is when one has several programs that compile from the same source. The source can be kept in one directory and each set of object files (along with a separate would be in a separate subdirectory. The "VPATH" macro would point to the source directory in this case.

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.c     gram.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. C. Johnson, "Yacc — Yet Another Compiler-Compiler", Bell Laboratories Computing Science Technical Report #32, July 1978.

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 $@
```

Report No. UIUCDCS-R-82-1081

NOTESFILE REFERENCE MANUAL
(abridged)

by

Raymond B. Essick IV
Rob Kolstad

February 14, 1983
(Revised: October 20, 1985)
(Printed: October 1, 1991)

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
1304 W. SPRINGFIELD AVENUE
URBANA, ILLINOIS 61801-2987

Supported in part by NASA Project NAS-1-138

TABLE OF CONTENTS

1 Introduction.

Notesfiles support computer managed discussion forums. Discussions can have many different purposes and scopes: the notesfile system has been designed to be flexible enough to handle differing requirements.

Each notesfile discusses a single topic. The depth of discussion within a notesfile is ideally held constant. While some users may require a general discussion of personal workstations, a different group may desire detailed discussions about the I/O bus structure of the WICAT 68000 (a particular workstation). These discussions might well be separated into two different notesfiles.

Each notesfile contains a list of logically independent notes (called base notes). A note is a block of text with a comment or question intended to be seen by members of the notesfile community. The note display shows the text, its creation time, its title, the notesfile's title, the author's name (some notesfiles allow anonymous notes), the number of "responses", and optionally a "director message". Each base note can have a number of "responses": replies, retorts, further comments, criticism, or related questions concerning the base note. Thus, a notesfile contains an ordered list of ordered lists. This arrangement has historically been more convenient than other proposals (e.g., trees were studied on the PLATO (trademark of Control Data Corporation) system).

The concept of a notesfile was originally implemented at the University of Illinois, Urbana-Champaign, on the PLATO system. The UNIX (trademark of Bell Laboratoris) notesfile system includes these ideas with adaptations and enhancements made possible by the UNIX environment.

The UNIX notesfile system was designed and implemented by Ray Essick at the University of Illinois, Urbana-Champaign. It provides users with the abilities to read notes and responses, write notes and responses, forward note text to other users (via mail) or other notesfiles, save note text in their own files, and sequence through a set of notesfiles seeing just new text. Each notesfile has a set of "directors" who manage the notesfile: they delete old notes, compress the file when needed, grant and restrict access to the notesfile, and set different notesfile parameters (e.g., title, "director message", policy note, whether notes' authors can be anonymous). Some notesfiles contain correspondence from other computers. Like the UNIX "USENET", notes and responses are exchanged (often over phone lines) with remote machines. The notesfile system provides automatic exchange and updating of notes in an arbitrarily connected network.

This document details the use of notesfiles from invocation through intersystem notes exchanges. The last chapter summarizes the entire set of commands for easy reference. An appendix contains detailed checklists for the installation of a notesfile system.

2 Using Notesfiles.

The notesfile system is invoked with a single command line. Most notesfile commands require only a single character (like the vi editor). Those that require more than one character are terminated by a carriage return.

2.1 Invocation.

Invoke the notesfile system with:

notes [-sxi] [-a subsequencer] [-t termtype] [-f nfile] [topic1] [topic2 ...]

The topic list (e.g., topic1) specifies the notesfiles to read. Invoking the notes system with NO arguments

yields a list of some available topics. When more than one topic is specified, the user encounters each topic sequentially (i.e., topic2 is entered upon completion of topic1).

The -s switch activates the "notesfile sequencer" which is discussed in section 2.8. Specify "-x" to use the extended sequencer. The "-i" flag selects yet another sequencing mode. The "-a" option specifies a particular subsequencer. This allows several users sharing a signon to maintain their own sequencing timestamp information.

The -t option directs the notesfile system to use "termtype" as the user's terminal type, overriding the TERM shell variable.

The -f option directs the notesfile system to read the contents of the file "nfile" for a list of notesfiles to read. See section 2.3 ("The -f Option") for more information on the format of this file.

2.2 Notesfile Names and Wildcards.

Notesfiles can be specified in several ways. The most common way is to merely give the name of the notesfile, such as "general". These notesfiles typically reside in the directory "/usr/spool/notes". Notesfiles may also be specified by their complete pathname; thus you could also refer to "general" by its full pathname "/usr/spool/notes/general". Using complete naming, notesfiles can be placed anywhere in the filesystem. This allows "private" notesfiles to be stored in personal directories.

The notesfile system supports pattern matching for names in the same manner as the shell. By using the shell meta-characters "*", "?", "[" and "]", the user can specify a number of notesfiles with a single entry. To read all the notesfiles that pertain to unix, enter the following line (the quotes are required to protect the metacharacters from interpretation by the shell):

```
notes "*unix*"
```

There are several ways to read the notesfiles test1, test2, test3 and test4:

```
notes test1 test2 test3 test4
notes "test?"
notes "test[1234]"
```

Entries can also be eliminated from the list of notesfiles to look at. By prefixing a notesfile name (possibly containing wildcard characters) with a '!', the notesfiles are excluded from the list to be examined. If one wished to look at all of the "test" notesfiles except test3, one could specify:

```
notes "test?" !test3
```

If you use the c shell, you will have to escape the '!', the history character:

```
notes "test?" \!test3
```

These features are available from the normal entry (notes) and the automatic sequencer entry (see section 2.8). Most notesfile programs recognize this format. Among those which do not are programs which must receive exactly one notesfile name.

2.3 The -f Option.

The "-f" option of the notesfile system specifies a file of notesfile names to read. The file consists of lines containing notesfile names:

```
nfgripes
net.unix-wizards
net.general
fa.telecom
```

The names start at the left margin; they are indented here for readability. Wildcard characters ("*", "?", "[", and "]") are acceptable in this context. Full names such as "/usr/spool/notes/general" are also accepted. Notesfiles can be eliminated through the "!" feature as described in section 2.2. The sequencer mode can be changed (see section 2.8) by inserting a line of the form:

```
-s
```

Again, this starts at the left margin. The "s" can be any of: "s", "x", "i", or "n". When a line of this form is read from the file, the sequencer mode is set to the corresponding mode: The normal "s"equencer, the e"x"tended sequencer, the "i"ndex sequencer, and "n"o sequencer.

To always enter nfgripes, micronotes, and bicycle while only entering the networked notesfiles "net.*" when new notes are present, one might use "notes -f myfile" with this "myfile":

```
-x
nfgripes
micronotes
bicycle
-s
net.*
```

2.4 General.

Almost all notesfile commands consist of exactly one character (no carriage return). Only commands that are longer than one character require a terminating carriage return (currently, choosing a note to read is the only non-single character command).

The commands were chosen to be easy to remember. Upper case forms of commands usually function like their lower case counterparts but with some additional feature or power (i.e., "w" writes a response, "W" includes the current displayed text in the response).

Some commands are available almost everywhere in the notesfile system. These include those for help, exiting, forking a shell, and making a comment for the suggestion box.

2.4.1 Help.

Typing "?" anywhere will list the available options in an abbreviated format.

2.4.2 Exiting.

Type "q" ("quit") to leave the current notesfile. Capital "Q" leaves the current notesfile and refrains from entering your last entry time into the sequencer table (see section "The Sequencer"). The notesfile system proceeds to the next topic in the invocation list. The "k" and "K" keys function exactly as "q" and "Q".

Use control-D ("signoff") to leave the notesfile system completely (without updating entry time information). The "z" command (which functions only when reading notes or responses or when on the index page) behaves similarly to control-D: the user exits the notesfile system immediately, but unlike control-D, updates the entry time information for the current notesfile.

2.4.3 Shells.

Fork a shell at any time by typing "!" (just like many other Unix programs).

2.4.4 Comments & Suggestions.

Type capital "B" ("suggestion Box") while on the index page or reading notes to make a comment or suggestion about the notesfile program. Your suggestion will be stored in another notesfile reviewed frequently by the notesfile system manager.

2.5 The Index Page.

When the notes system is invoked without the -s option, the user sees an index of the most recent notes. A sample page is shown below:

Workstation Discussion 2:03 pm Jan 4, 1982

12/9/81	2 Stanford SUN	4	horton
	3*WICAT 68000		kolstad
	4 M68000	1	horton
	5 Dolphin	3	duke!johnson
12/10	6 CDC Standalone	1	smith
	8 IBM Personal Computer		henry
	9 Personal computers harmful?	8	Anonymous
	10 Ethernet interfaces 3 mhz?	23	essick
	11 Requirements for uiucdcs	10	botten
1/1/82	12 Happy New Year!	5	mjk

The upper left corner shows the notesfile's title. In this example, the notesfile discusses personal workstations. The current time and date are displayed in the upper right corner. Approximately ten note titles are displayed (if available). More notes are displayed on longer screens (such as the Ann Arbor Ambassador). Each note is displayed with its date (if different from the previous date), note number, title, number of responses (if any), and author. The first note above was written by user "horton" on December 9th, is entitled "Stanford SUN" and has four responses. Note 7 has been deleted for some reason (by either its author or a notesfile director). Note 5 was written by user "johnson" whose signon resides on the "duke" system. Note 9 was written by an author who preferred to remain unidentified. Notes with director messages (sometimes denoting importance) are displayed with a "*" next to the note number (see note 3 above).

From the index page the user may:

- Scroll the index forward or backward.
- Read a note.
- Write a note.
- Go to the next unread note.
- Search for notes or responses after a specific date/time.

- Search for keywords within notes' titles.
- Search for notes/responses by a specific author.
- Go to another notesfile.
- Consult the notesfile's archive.
- Read the policy note.
- Check on anonymous and networked status.
- Register a complaint/suggestion about notesfiles.
- Fork a shell.
- Exit the notes program.
- Invoke notesfile director options (if the user is a director).

2.5.1 Scrolling the Index Page.

Scroll the index page by:

+, <return>, <space> forward one page
* forward to the most recent page (* is multiple +'s)
\- backward one page
= backward all the way (= is multiple -'s)

2.5.2 Choosing Notes & Responses.

While on the index page, choose a note to read by typing its number followed by a carriage return. (This is the only command that requires a carriage return after it.) Usually the space bar is used to scan text. To skip to a particular note or response, use the features below.

While reading a note, ";" or "+" advances to the first response of the note. The next note is displayed if there are no responses. The number keys ("1", "2", ... , "9") advance that many responses. If there are fewer responses, the last response is displayed. The return key skips the responses and goes to the next note. Press "-" or backspace to see the previous page of the current note; if the page currently displayed is the first, the notesfile program displays the first page of the previous note.

While a response is on the screen, the ";" and "+" keys display the next response. As with reading a note, if there are no further responses these keys advance to the next note. The number keys ("1", ... , "9") will advance the appropriate number of responses. If there are fewer responses, the last response is displayed. The "-" or backspace keys display the previous page of the current response. If the current page is the first page of the response, these keys display the first page of the previous response. Enter "=" to see the base note of the current note string. Press the return key to proceed to the next note.

2.6 Notes & Responses.

2.6.1 Reading Notes.

After selecting a note from the index page (or entering the notesfile with your "sequencer" on), the note is displayed. A sample display is shown below:

Note 15 Workstation Discussion 2 responses
horton WICAT 150 4:03 pm Dec 11, 1981

Wicat System 150

8 MHz 68000, Mem. mgmt, Multibus architecture, 256k to 1.5 Mb RAM,16/32/64Kbyte EPROM,
10 ms interval timer, 2 RS232 (19.6k async, 56k sync), 16 bit parallel intelligent disk controller,
10 Mbyte winchester (5.25", 3600 rpm, access: 3 ms trk-trk, 70 avg, 150 max),

960Kb floppy (5.25", 300 rpm, access 10 ms trk-trk, 267 avg, 583 max)
Options: battery backed clock, graphics with touch panel, video disk control,
High Speed Serial Network Interface
Unix/V7 avail, Pascal, C, APL, ADA, Cobol, Fortran, Lisp, Basic, Asm

This is note number 15 in the "Workstation Discussion" file. User "horton" wrote this note at 4:03 pm on December 11th, 1981. Two responses have been written. The note's title is "WICAT 150". If a director had written the note, the "director message" might have been displayed beneath the note's title. Director's notes sometimes contain important information or new policies.

Since notes and responses can each be up to 3 Mbytes long, the display routine breaks text into pages automatically. For all but the last page of a long note or response, the lower right corner of the display shows the percentage of the note that has been shown. For all but the first page of long text, the message "[Continued]" appears in the upper left portion of the display. Use the space bar to see the next page of a long note or response. When the last page is displayed, the space key functions as the ";" key: it proceeds to the next response. The "-" and backspace keys back up the display to the previous page. Only the first 50 pages of text are managed this way; typing "-" from the fifty-second page will return to the fiftieth page. The "=" key returns to the first page of the note.

While reading a note, it is possible to:

- Display the next, previous, or first page of the note.
- Write a response to the displayed note.
- Read next note or previous note.
- Read next unread response or note.
- Return to the index page.
- Skip to a given response.
- Delete the note (if you are its author or a file director).
- Edit the note's title (if it is yours).
- Edit the note (if it is yours and there are no responses).
- Copy the note to another notesfile.
- Save the note in your file space.
- Mail the note to someone.
- Talk ("write") to the author of the note.
- Search for keywords in note titles.
- Search for notes/responses by a particular author.
- Toggle the director message (if privileged).
- Fork a shell.
- Go to another notesfile.
- Make a comment or suggestion about notesfiles.
- Exit the notesfile program.

2.6.2 Reading Responses.

Response displays are similar to those of main notes with the exception that "Response x of y" replaces the note's title. The first response to note 15 is shown below:

Note 15 Workstation Discussion
koehler Response 1 of 2 11:53 pm Dec 11, 1981

Does anyone have any insight about the relative speeds of the Winchester disks available on these systems? The previous disk seems to have track to track response times commensurate with reasonably fast 8" floppies. I wonder if some of the manufacturers are using disks that will not meet reasonable specifications for response time for these kinds of applications.

On the other hand, with intelligent layout of file sectors, the I/O system could romp and stomp on often used files...

==========

The commands for manipulating the text of a long response are the same as those for looking at long notes. Typing space will move to the next page. Typing "-" or backspace will display the previous page, within the same limitations as for reading notes (only 50 pages are kept). Press "=" to go back to the first page of the text.

The options available while reading responses include:

- Display the next, previous, or first page of the response.
- Go to a different response (usually the next one).
- Go to the next unread note/response.
- Reread the base note.
- Reread the previous note.
- Return to the index page.
- Copy the response to another notesfile.
- Mail the response to someone.
- Save the response in your file space.
- Talk to the response's author.
- Write another response to the note.
- Search for keywords in note titles.
- Search for notes/responses by particular authors.
- Delete the response (if you are its author or a file director).
- Edit the response (if it is yours and there are no later responses).
- Fork a shell
- Go to another notesfile.
- Register a suggestion or complaint about the notesfile program.
- Exit the notesfile program.

2.6.3 Writing Notes & Responses.

Write new base notes by hitting "w" while reading the index page. The notesfile system will then invoke an editor ("ed" by default; use either of the shell variables NFED or EDITOR to change it). After the prompt, compose the text you wish to enter, then write the text to the disk and leave the editor. The system will prompt you for various options if they are available: anonymity, director message status, and the note's title.

To write a response to a note type "w" while that note or any of its responses is displayed. The same steps used to write a base note should then be followed.

2.6.4 Mailing Notesfile Text.

Both notes and responses can be mailed to other users (with optional appended text). The capital "M" ("mail") command gives you the opportunity to edit the text then send it to anyone. Its inferior counterpart, "m", allows you to mail a message to anyone. To mail to the author of the text, use capital "P" ("Personal comment") to send the text and your comments; use "p" for a simple letter.

To use a specific mail program, set the environment variable MAILER. If this is not set, a standard mail program is used.

2.6.5 Forwarding Text To Other Notesfiles.

There are several methods for forwarding text from one notesfile to another. Single notes or responses can be copied with the "c" or "C" command while entire note strings can be forwarded with the "f" and "F" commands.

The "f" ("forward") command is given when a base note is displayed on the screen. When given, the "f" command causes the base note and all of its responses to be copied to another notesfile. The user is prompted for the destination notesfile. The copied note and all of the copied responses contain header information detailing their origin. Where "f" copies the note string without change, the "F" command allows the user to edit the text of the note and each response before inserting it into the target notesfile.

The "c" ("copy") command prompts for a destination notesfile then copies the currently displayed note or response to the target notesfile. The user is allowed to choose between forwarding the note as a response or as a new base note. The "c" command does not give the user a chance to edit the text before inserting it in the new notesfile. The extended copying command "C" allows editing of the note text before it is copied to the other notesfile.

Both the "c" and "C" commands provide for the forwarded text to be entered as either a new note or as a response to an existing note. In the latter case, an index page is given to the user for choosing the appropriate note to which to respond.

2.6.6 Saving Text in Local Files.

The "s" ("save") command appends the current displayed text to a file of your choice (which is created if not present). Notesfiles prompts for the file name; typing only a carriage return aborts the command -- no text is saved. Capital "S" appends the base note and all its responses. The number of lines saved and the name of the file written are printed when the command completes.

2.6.7 Deletion.

Capital "D" ("delete") deletes a note or response if it is yours and has no subsequent responses. Notes already sent to the network can not be deleted by non-directors. Directors can delete any note or response with the "Z" ("zap") command.

2.6.8 Online Communication.

Typing "t" ("talk") attempts to page the author of the current displayed text. The Unix "write" command to him/her is issued if the author is local and non-anonymous. If the environment variable WRITE is defined, the program it specifies is used to write to the author.

2.6.9 Editing Note Titles.

While reading a base note, type "e" ("edit") to change the note's title (provided you are the author of the note or a notesfile director).

2.6.10 Editing Notes/Responses.

"E" allows editing of the text of a note or response. It is not permitted to edit an article if it has subsequent responses or if it has been sent to the network. If the "later responses" are deleted, it is possible to edit the original text.

2.7 Other Commands.

2.7.1 Returning to the Index Page.

Type "i" ("index") while reading notes or responses to return to the index page.

2.7.2 Searching Titles for Keywords.

While reading, you can search backwards for keywords appearing in note titles. Typing "x" ("x is the unknown title") prompts for the substring to be found. Searching begins at the current note (or from the last note shown on the index page) and proceeds towards note 1. The search is insensitive to upper/lowercase distinctions. Use upper case "X" to continue the search. The search can be aborted by hitting the RUBOUT (or DELETE) key.

2.7.3 Searching for Authors.

The "a" command searches backwards for notes or responses written by a specific author. Notesfiles prompts for the author's name. The "A" command continues the search backwards. The author name may be preceded by an optional 'system!'. Abort the search by hitting the RUBOUT (or DELETE) key.

The entire name need not be specified when searching for articles by a particular author. Author searching uses substring searching. Searching for the author "john" will yield articles written by a local user "john", a remote user "somewhere!johnston", and any articles from the "uiucjohnny" machine. Author searching is case sensitive.

2.7.4 Stacking Notesfiles.

Sometimes it is useful to be able to glance at another notesfile while reading notes. Using "n", the user can save (stack) his current place and peruse another notesfile.

When on the index page or while reading notes/responses, type "n" ("nest") to read another notesfile. Notesfiles prompts for the notesfile to read. If the notesfile exists, the place is marked in the old notesfile and the new one's index is displayed.

Type any of the standard keys to leave the nested notesfile. Both "q" and "Q" leave the nested notesfile and return to the previously stacked notesfile. Control-d ("signoff") causes the notesfile program to exit regardless of the depth of nesting.

Sequencing is turned off in the new notesfile regardless of its state in the old notesfile. The depth of the stack of notesfiles is limited only by the amount of memory available to the user.

2.7.5 Accessing Archives.

As notesfiles grow, it becomes impractical to keep every discussion. In some cases, the old discussions are deleted; other cases require these old discussions to be saved somewhere. Each active notesfile can have an archive notesfile. An archive notesfile contains the old discussions from the active notesfile.

The archive of an active notesfile is accessed by explicitly naming the notesfile (/usr/spool/oldnotes/micronotes for example) or through the "N" command from the active notesfile.

2.7.6 Policy Note.

A notesfile director can write an optional policy note to describe the purpose of a notesfile. Read the policy note by typing "p" ("policy") from the index page.

2.8 The Sequencer.

Most users prefer to scan notesfiles and see only those notes written since their last reading. The notesfile "sequencer" provides this capability. It is activated by the "-s" option ("sequencer") on the command line. When the sequencer is activated, the notesfile system automatically remembers the last time the user read notes in each notesfile. Subsequent entries to the notesfile can use the "last time" information to show only new notes and responses. If there is nothing new in a notesfile, the sequencer proceeds to the next notesfile specified in the command line.

The normal sequencer does not give the user a chance to read the notesfile if there are no new notes or responses; sometimes it is desirable to be able to do so. Use the "-x" option to enable the sequencer and enter the notesfile even if there are no new notes.

No keys need be pressed if there are no new notes in the entire list and the normal ("-s") sequencer mode is selected. With the extended ("-x") sequencer, the user must type "q", "Q", or control-d for each notesfile regardless of whether there are new notes.

The "-i" mode of sequencing is similar to the "-s" mode. Using the "-i" mode, notesfiles without new entries are passed over. The user starts reading on the index page of notesfiles which contain new notes.

2.8.1 Seeing New Notes and Responses.

The sequencer always shows the base note of a modified note string, whether or not is has been shown before, in order to establish the context of the new response(s). The "j" command skips to the next modified text (note or response).

If the rest of a particular note string seems uninteresting, skip to the next modified note string with the "J" ("big Jump") command. This skips any new responses on the current note string. It is common to follow closely only a few note strings, skipping others using the "J" command.

The "last time" information is kept in a special file for each user. When the sequencer is enabled, the time for the notesfile is loaded into a variable and used to specify which notes and responses are new. If the sequencer is not enabled, this variable is initialized to January 1, 1970. The "j" and "J" keys use this variable to determine which notes and responses are "new".

If the sequencer is enabled, after exiting a notesfile the "last time" information is updated to the time that the user entered this notesfile. The entry time is used rather than the exit time to ensure that all notes are seen, including ones written during the just completed session. If the sequencer is disabled, the "last time" information is not modified. The "last time" information for a particular notesfile is updated as that notesfile is exited; using "Q" or control-D later will have no effect on the sequencer information for notesfiles already read.

The "o" and "O" commands allow the user to modify the variable used to determine whether notes and responses are "new". The "o" command allows the user to set this variable to any date he wishes. Use the "O" command to set this variable to show only notes and responses written that day. The "last time" file kept for each user is never modified by the "o" and "O" commands.

When no more new notes or responses exist, both the "j" and "J" commands will take the user to

the index page. To exit the notesfile, use the "q" command. Exiting with "q" will update the user's "last entry" time. Exiting with capital "Q" will NOT modify the "last entry" time for that notesfile (neither will control-D).

The "l" and "L" command behave similarly to "j" and "J". The difference is that while "j" and "J' take the user to the last index page when no more new notes or responses exist, the "l" and "L" commands will leave the notesfile as if a "q" had been typed. Thus when no more new notes exist, the "l" command is like typing "jq".

2.8.2 Alternate Sequencers.

If several people share a login account, it is convenient for each to have a set of sequencing timestamps. This is accomplished through the use of the subsequencer option of notesfiles.

Specifying the -a option and a subsequencer name causes notes to use a different sequencing timestamp file. Many different subsequencer names can be used with each login account.

The main sequencer file for a given account is distinct from each of its subsequencer files. Each of the subsequencer files is normally distinct. If the subsequencer names are not unique in their first 6 characters, subsequencer files may collide.

2.8.3 Automatic Sequencing.

An alternate entry to the notes program allows the user to invoke notes with the sequencer enabled and a list of notesfiles to be scanned with a single, simple command. The "autoseq" command is invoked by typing

```
autoseq
```

and reads the environment variable "NFSEQ" to find the names of all notesfiles to be scanned. On some systems, the "autoseq" command may be known as "readnotes", "autonotes" or some similar variant; substitute the appropriate name in the following paragraphs. The "NFSEQ" variable should be defined in .profile for Bourne shell users as follows:

```
NFSEQ="pbnotes,micronotes,helpnotes,works"
export NFSEQ
```

For users of the C shell, the following line should be added to the .login file:

```
setenv NFSEQ "pbnotes,micronotes,helpnotes,works"
```

With NFSEQ assigned this value, a call to autoseq will process the notesfiles "pbnotes", "micronotes", "helpnotes", and "works" with the sequencer turned on.

The full naming conventions, pattern matching capabilities, and '!' exclusion described in section 2.2 ("Notesfile Names and Wildcards") are available in autoseq. To read all notesfiles with "unix" in their names, and the four test notesfiles ("test1" though "test4"), the NFSEQ variable might be defined as:

```
NFSEQ="*unix*,test[1234]"
```

If the first character of an entry in the NFSEQ list is ":", the notesfile system reads the file name following for a list of notesfiles. To have the automatic sequencer read the file "/usr/essick/.nfseq" for a list of notesfiles to scan, define NFSEQ as:

NFSEQ=":/usr/essick/.nfseq"

For this feature to work, the file must have group read privileges. The notesfile program runs "set-uid" and can not read files which are readable only by the owner.

The following definitions are also valid. The first one reads the notesfiles specified in the file "/usr/essick/.nfseq" and then reads the notesfiles pbnotes and micronotes. The second definition will read the notesfile pbnotes, those specified in "/usr/essick/.nfseq", micronotes and the ones specified in "/usr/essick/.other". If the notesfile program is unable to read the file specified, it skips to the next entry. For a description of the format of these files, see the section 2.3, "The -f Option".

NFSEQ=":/usr/essick/.nfseq,pbnotes,micronotes"

NFSEQ="pbnotes,:/usr/essick/.nfseq,micronotes,:/usr/essick/.other"

The automatic sequencer uses the "-s" mode of sequencing. The user does not enter notesfiles which have no new text. By specifying "-x" or "-i" on the command line, the user can use the appropriate sequencer mode.

The subsequencer option of notes is available from the autoseq program by specifying "-a name" on the command line, and has identical semantics with use of this option when invoking notes.

2.9 Environment Variables.

The notesfile program reads several environment variables to tailor the system to the user's preferences. Below is a list of the variables, their purpose, and their default values. These defaults are for UNIX 4.xBSD and may be slightly different for other versions of UNIX.

- "NFED" specifies which editor will be invoked when the user writes a note or response. If this variable is not specified, the notesfile system looks for the environment variable "EDITOR" (which many other programs use). If neither "NFED" nor "EDITOR" are defined, a default editor is used (/bin/ed).
- "NFSEQ" is a list of notesfiles that the user wishes to scan using the automatic sequencing entry to notesfiles. The use of this variable is described in the section on sequencing. If unspecified, the system uses a standard set which usually includes "general" and "net.general".
- "PAGER" is the paging program ("more", "pg") which is used for scrolling the help files. The default paging program is /usr/ucb/more.
- "MAILER" determines the mail program to use. This defaults to /usr/ucb/mail.
- "WRITE" is used to specify the program for communication between users. If undefined, the Unix program "write" is used.
- "TERM" determines the type of terminal in use. This must be set for notes to know what screen handling conventions to use. In most cases the value will be correctly initialized by the system at login time.
- "SHELL" specifies which shell the user is running. This will almost always be set by the operating system.

3 Other Notesfile Utilities.

The notesfile distribution includes utility programs to provide hard copy output, additional interfaces to user programs, and statistics. They are described below.

3.1 Hard Copy Output.

The program "nfprint" sends to standard output a nicely formatted listing of the notesfile in its command line. Its format is:

nfprint [-lnn] [-p] [-t] topic [note#] [note#-note#] [...]

The "-l" option specifies an alternate page size (the default is 66). The optional note number list specifies that only certain notes of the notesfile are to be printed. The list can specify individual notes and ranges. The notes are printed in the order specified.

The -p option specifies that each notestring is to begin on a new page. The -t option signifies that only a table of contents is to be generated.

3.2 Piped Insertion of Notes.

The nfpipe program enters text from the standard input into a notesfile:

nfpipe topic [-t title] [-d] [-a]

The -t option allows specification of a title. The -d and -a options specify the director and anonymous flags respectively (if available). If no title is specified, one is manufactured from the first line of the note.

3.3 User Subroutines.

3.3.1 Nfcomment.

The nfcomment subroutine is callable from a user's C program. It allows any user program to enter text into a notesfile:

nfcomment (nfname, text, title, dirflag, anonflag)

The parameters are:

```
char *nfname;   /* name of notesfile */
char *text;     /* null terminated text to be entered */
char *title;    /* if non-null, title of note */
int dirflag;    /* != 0 -> director flag on (if allowed) */
int anonflag;   /* != 0 -> anonymous note (if allowed) */
```

If the text pointer is NULL, the text of the note will be read from standard input. If no title is specified the subroutine will manufacture a title from the first line of the note. This routine is useful for error reports, user comments about programs, and automatic logging of statistics or internal states.

This routine can be loaded with a C program by specifying '-lnfcom' on the 'cc' command line.

3.3.2 Nfabort.

Nfabort allows users to generate core images of their process, save the core image in a "known" place, and log that fact in a notesfile. This proves useful for intermittent failures; The programmer regularly scans the notesfile and can examine the core dump at leisure. Some of the problems of recreating conditions which cause errors are eliminated by this approach.

Nfabort is callable from the user program. It accepts the following parameters:

nfabort (nfname, message, title, cname, exitcode)

The parameters are:

```
char *nfname;   /* name of notesfile */
char *message;  /* text string to insert */
char *title;    /* title of the message */
char *cname;    /* prefix for core image destination */
int exitcode;   /* code for exit() */
```

The core image is placed in the file specified by concatenating the "cname" argument and a unique integer (the process id of the current process). The notesfile specified by the "nfname" parameter receives a note whose body consists of the text pointed to by "message" and a line telling the complete pathname of the core image. The title of the note is specified by the "title" parameter. After the core image is generated and the note has been written, nfabort terminates with the exit code specified by the "exitcode" parameter.

Nfabort generates default values for each of the string parameters if NULL pointers are passed. This routine can be loaded with a C program by specifying '-lnfcom' on the 'cc' command line.

3.4 Statistics.

The notesfile system keeps statistics on where notes and responses originate, the number of network accesses, duplications and orphaned responses. Combined with the use of the log maintained by the notesfile networking software, monitoring notesfile traffic is quite easy.

The -s option specifies that only a summary is to be produced, skipping the individual reports. Wildcard constructs with '*', '?', '[', and ']' are recognized by nfstats. Invoke the statistics program with:

nfstats [-s] topic1 [...]

Typical output is:

rbenotes on uiucdcs at 6:24 pm May 7, 1982

	NOTES	RESPS	TOTALS
Local Reads	359	115	474
Local Written	53	55	108
Networked in	0	0	0
Networked out	0	0	0
Network Dropped	0	0	0

Network Transmissions: 0 Network Receptions: 0
Orphaned Responses Received: 0 Entries into notesfile: 109
Total time in notesfile: 66.57 minutes Average Time/entry: 0.61 minutes
Created at 10:04 pm May 5, 1982, Used on 3 days

A combined set of statistics is produced at the end of listings of more than one notesfile. The statistics are largely self explanatory.

3.5 Checking for New Notes.

The checknotes program checks the notesfiles specified by the NFSEQ environment variable to determine if there are new notes. The exit code is arranged to make the program useful in shell scripts: 0 (TRUE) is there are new notes, 1 (FALSE) otherwise.

Use the "-q" option to receive a message

There are new notes

if one or more of the notesfiles have notes/responses written since the user's last entry time into that notesfile.

The "-n" option is similar to the "-q" option, with the exception that it yields output when there are no new notes. The output of checknotes with the "-n" option is:

There are no new notes

Use "-v" to print the name of each notesfile with new notes/responses. The "-s" option is suitable for use in conditional expressions in shell scripts; no output is generated by this option.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **termcap**(5) database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Revised 16 April 1986

Contents

Appendixes

1. Overview

In making available the generalized terminal descriptions in **termcap**(5), much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called ***terminal screen***. The package's idea of what the terminal's screen currently looks like, *i.e.*, what the user sees now. This is a special *screen*:

screen: This is a subset of windows which are as large as the terminal screen, *i.e.*, they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source. The header file **<curses.h>** needs to include **<sgtty.h>**, so the one should not do so oneself[1]. Also, compilations should have the following form:

cc [*flags*] file ... **–lcurses –ltermcap**

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the

[1] The screen package also uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. It is redundant (but harmless) for the programmer to do it, too.

terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does **not** change the terminal.* Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this", and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided[2]. This convention of prepending function names with a "**w**" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "**mv**" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

---

[2] Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class "register" (*e.g., reg int i;*)
bool	boolean type, actually a "char" (*e.g., bool doneit;*)
TRUE	boolean "true" flag (1).
FALSE	boolean "false" flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must ***always*** be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *cbreak()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a char-

acter at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, *i.e.*, printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, *i.e.*, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routines *touchwin()*, *touchline()*, and *touchoverlap()* are provided to make it look like a desired part of window has been changed, thus forcing *refresh()* check that whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *getttmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to initscr, *endwin()* should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as **rogue** and **vi**[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"[4] and optimizing **more**(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the

[3] **rogue** actually uses these functions, **vi** does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as **rain**, **rocket**, and **gun**.

lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The **termcap**(5) database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, *HO* is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by **gtty**(2) and **stty**(2) The second, *setterm()*, a larger task by reading in the descriptions from the **termcap**(5) database. This is the way these routines are used by *initscr()*:

```
if (isatty(0)) {
        gettmode();
        if ((sp=getenv("TERM")) != NULL)
                setterm(sp);
                 else
                        setterm(Def_term);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[7]. If it is, *gettmode()* sets the terminal description modes from a **gtty**(2) *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from **termcap**(5) associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def_term* is used. The *TI* and *VS* sequences initialize the terminal (*_puts()* is a macro which uses *tputs()* (see **termcap**(3)) and *_putchar()* to put out a string). *endwin()* undoes these things.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor **vi** uses many of these features, and the routines it uses to do this

---

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

[6] These names are identical to those variables used in the **termcap**(5) database to describe each capability. See Appendix A for a complete list of those read, and the **termcap**(5) manual page for a full description.

[7] *isatty()* is defined in the default C library function routines. It does a **gtty**(2) on the descriptor and checks the return value.

[8] Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto()* from the **termlib**(7) routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

5. The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as it's "**w**" counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

5.1. Output Functions

addch(ch) †
char ch;

waddch(win, ch)
*WINDOW *win;*
char ch;

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

addstr(str) †
*char *str;*

waddstr(win, str)
*WINDOW *win;*
*char *str;*

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

box(win, vert, hor)
*WINDOW *win;*

char vert, hor;

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() †

wclear(win)
*WINDOW *win;*

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf) †
*WINDOW *scr;*
bool boolf;

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

clrtobot() †

wclrtobot(win)
*WINDOW *win;*

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

clrtoeol() †

wclrtoeol(win)
*WINDOW *win;*

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

delch()

wdelch(win)
*WINDOW *win;*

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
*WINDOW *win;*

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() †

werase(win)
*WINDOW *win;*

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "**mv**" command.

flushok(win, boolf) †
*WINDOW *win;*
bool boolf;

Normally, *refresh()* *fflush()*'s *stdout* when it is finished. *flushok()* allows you to control this. if *boolf* is TRUE (*i.e.*, non-zero) it will do the *fflush()*; if it is FALSE. it will not.

idlok(win, boolf)
*WINDOW *win;*
bool boolf;

Reserved for future use. This will eventually signal to *refresh()* that it is all right to use the insert and delete line sequences when updating the window.

insch(c)
char c;

winsch(win, c)
*WINDOW *win;*
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

insertln()

winsertln(win)
*WINDOW *win;*

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged.

move(y, x) †
int y, x;

wmove(win, y, x)
*WINDOW *win;*
int y, x;

Change the current (y, x) co-ordinates of the window to (*y, x*). This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
*WINDOW *win1, *win2;*

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
*WINDOW *win1, *win2;*

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, *i.e.*, blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
*char *fmt;*

wprintw(win, fmt, arg1, arg2, ...)
*WINDOW *win;*
*char *fmt;*

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() †

wrefresh(win)
*WINDOW *win;*

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

As a special case, if *wrefresh()* is called with the window *curscr* the screen is cleared and repainted as it is currently. This is very useful for allowing the redrawing of the screen when the user has garbage dumped on his terminal.

standout() †

wstandout(win)
*WINDOW *win;*

standend() †

wstandend(win)
*WINDOW *win;*

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

5.2. Input Functions

cbreak() †

nocbreak() †

crmode() †

nocrmode() †

Set or unset the terminal to/from cbreak mode. The misnamed macros *crmode()* and *nocrmode()* are retained for backwards compatibility with ealier versions of the library.

echo() †

noecho() †

Sets the terminal to echo or not echo characters.

getch() †

wgetch(win)
*WINDOW *win;*

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str) †
*char *str;*

wgetstr(win, str)
WINDOW **win;*
char **str;*

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

_putchar(c)
char *c;*

Put out a character using the *putchar()* macro. This function is used to output every character that **curses** generates. Thus, it can be redefined by the user who wants to do non-standard things with the output. It is named with an initial "_" because it usually should be invisible to the programmer.

raw() †

noraw() †

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see *nl()*).

scanw(fmt, arg1, arg2, ...)
char **fmt;*

wscanw(win, fmt, arg1, arg2, ...)
WINDOW **win;*
char **fmt;*

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

baudrate() †

Returns the output baud rate of the terminal. This is a system dependent constant (defined in **<sys/tty.h>** on BSD systems, which is included by **<curses.h>**).

delwin(win)
WINDOW **win;*

[9] UNIX is a trademark of Bell Laboratories.

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin()* allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before *initscr()* (or *gettmode()* and *setterm()*) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

erasechar() †

Returns the erase character for the terminal, *i.e.*, the character used by the user to erase a single character from the input.

*char **
getcap(str)
char **str;*

Return a pointer to the **termcap** capability described by *str* (see **termcap**(5) for details).

getyx(win, y, x) †
WINDOW **win;*
int *y, x;*

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

inch() †

winch(win) †
WINDOW **win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initialy "dumb"). If the boolean *My_term* is true, *Def_term* is always used. If the system supports the **TIOCGWINSZ** *ioctl(2)* call, it is used to get the number of lines and columns for the terminal, otherwise it is taken from the **termcap** description.

killchar() †

Returns the line kill character for the terminal, *i.e.*, the character used by the user to erase an entire line from the input.

leaveok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
char **termbuf, *name;*

fullname(termbuf, name)
char **termbuf, *name;*

longname() fills in *name* with the long name of the terminal described by the **termcap** entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *termbuf* is usually set via the termlib routine *tgetent()*. *fullname()* is the same as *longname()*, except that it gives the fullest name given in the entry, which can be quite verbose.

mvwin(win, y, x)
WINDOW **win;*
int *y, x;*

Move the home position of the window *win* from its current starting coordinates to (*y, x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything. For subwindows, *mvwin()* also returns ERR if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

WINDOW *
newwin(lines, cols, begin_y, begin_x)
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

nl() †

nonl() †

Set or unset the terminal to/from nl mode, *i.e.*, start/stop the system from mapping **<RETURN>** to **<LINE-FEED>**. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) †
WINDOW **win;*
bool *boolf;*

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchline(win, y, startx, endx)
WINDOW **win;*
int *y, startx, endx;*

This function performs a function similar to *touchwin()* on a single line. It marks the first change for the given line to be *startx*, if it is before the current first change mark, and the last change mark is set to be *endx* if it is currently less than *endx*.

touchoverlap(win1, win2)
WINDOW **win1, *win2;*

Touch the window *win2* in the area which overlaps with *win1*. If they do not overlap, no changes are made.

touchwin(win)
WINDOW **win;*

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

*WINDOW **
subwin(win, lines, cols, begin_y, begin_x)
WINDOW **win;*
int *lines, cols, begin_y, begin_x;*

Create a new window with *lines* lines and *cols* columns starting at position (*begin_y, begin_x*) inside the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin_y, begin_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin_y*) or (*COLS* - *begin_x*) respectively.

unctrl(ch) †
char *ch;*

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you may have to have **#include <unctrl.h>** in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by *initscr()*.

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
*WINDOW *win;*

Scroll the window upward one line. This is normally not used by the user.

savetty() †

resetty() †

savetty() saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

setterm(name)
*char *name;*

Set the terminal characteristics to be those of the terminal named *name*, getting the terminal size from the **TIOCGWINSZ** *ioctl(2)* if it exists, otherwise from the environment. This is normally called by *initscr()*.

tstp()

If the new **tty**(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see **termcap**(5).

1.2. Overview

Capabilities from **termcap** are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, i e.g. , **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

1.3. Variables Set By setterm()

variables set by *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE

variables set by *setterm()*

Type	Name	Pad	Description
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with ***X*** are reserved for severely nauseous glitches

For purposes of *standout()*, if *SG()* is not 0, *SO()* is set to *NULL()*, and if *UG()* is not *0()*, *US()* is set to *NULL()*. If, after this, *SO()* is *NULL()*, and *US()* is not, *SO()* is set to be *US()*, and *SE()* is set to be *UE()*.

1.4. Variables Set By gettmode()

variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

1. The WINDOW structure

The WINDOW structure is defined as follows:

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 *
 *      @(#)win_st.c        6.1 (Berkeley) 4/24/86";
 */

# define        WINDOW struct _win_st

struct _win_st {
        short           _cury, _curx;
        short           _maxy, _maxx;
        short           _begy, _begx;
        short           _flags;
        short           _ch_off;
        bool            _clear;
        bool            _leave;
        bool            _scroll;
        char            **_y;
        short           *_firstch;
        short           *_lastch;
        struct _win_st  *_nextp, *_orig;
};

# define        _ENDLINE        001
# define        _FULLWIN        002
# define        _SCROLLWIN      004
# define        _FLUSH          010
# define        _FULLLINE       020
# define        _IDLINE         040
# define        _STANDOUT       0200
# define        _NOCHANGE       -1
```

_cury and *_curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *_maxy* and *_maxx* are the maximum values allowed for (*_cury*, *_curx*). *_begy* and *_begx* are the starting (y, x) co-ordinates on the terminal for the window, *i.e.*, the window's home. *_cury*, *_curx*, *_maxy*, and *_maxx* are measured relative to (*_begy*, *_begx*), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next *refresh()* call. This is only meaningful for screens. The initial clear-screen for the first *refresh()* call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *_leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *_scroll* is TRUE if scrolling is allowed.

---

[10] All variables not normally accessed directly by the user are named with an initial "_" to avoid conflicts with the user's variables.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the *i*th line, and

```
_y[i][j]
```

is the *j*th character on the *i*th line. *_flags* can have one or more values or'd into it.

For windows that are not subwindows, *_orig* is NULL . For subwindows, it points to the main window to which the window is subsidiary. *_nextp* is a pointer in a circularly linked list of all the windows which are subwindows of the same main window, plus the main window itself.

_firstch and *_lastch* are *malloc()*ed arrays which contain the index of the first and last changed characters on the line. *_ch_off* is the x offset for the window in the *_firstch* and *_lastch* arrays for this window. For main windows, this is always 0; for subwindows it is the difference between the starting point of the main window and that of the subindow, so that change markers can be set relative to the main window. This makes these markers global in scope.

All subwindows share the appropriate portions of *_y*, *_firstch*, *_lastch*, and *_insdel* with their main window.

_ENDLINE says that the end of the line for this window is also the end of a screen. **_FULLWIN** says that this window is a screen. **_SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **_FULLLINE** says that the width of a line is the same as the width of the terminal. If **_FLUSH** is set, it says that *fflush(stdout)* should be called at the end of each *refresh()* **_STANDOUT** says that all characters added to the screen are in standout mode. **_INSDEL** is reserved for future use, and is set by *idlok()*. *_firstch* is set to **_NOCHANGE** for lines on which there has been no change since the last *refresh()*.

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)twinkle1.c          6.1 (Berkeley) 4/24/86";
#endif not lint

# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

typedef struct {
        int     y, x;
} LOCS;

LOCS    Layout[NCOLS * NLINES];     /* current board layout */

int     Pattern,                    /* current pattern number */
        Numstars;                   /* number of stars in pattern */

char    *getenv();

int     die();

main()
```

```
{
        srand(getpid());                        /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);

        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on '*'s */
                puton(' ');                     /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die()
{
        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS - 1, LINES - 1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard()
{
        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp->x = x;
                                lp++;
                        }
        Numstars = lp - Layout;
}
```

```
/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int     y, x; {

            switch (Pattern) {
              case 0:           /* alternating lines */
                    return !(y & 01);
              case 1:           /* box */
                    if (x >= LINES && y >= NCOLS)
                          return FALSE;
                    if (y < 3 || y >= NLINES - 3)
                          return TRUE;
                    return (x < 3 || x >= NCOLS - 3);
              case 2:           /* holy pattern! */
                    return ((x + y) & 01);
              case 3:           /* bar across center */
                    return (y >= 9 && y <= 15);
            }
            /* NOTREACHED */
}

puton(ch)
reg char            ch;
{
            reg LOCS            *lp;
            reg int             r;
            reg LOCS            *end;
            LOCS                temp;

            end = &Layout[Numstars];
            for (lp = Layout; lp < end; lp++) {
                    r = rand() % Numstars;
                    temp = *lp;
                    *lp = Layout[r];
                    Layout[r] = temp;
            }

            for (lp = Layout; lp < end; lp++) {
                    mvaddch(lp->y, lp->x, ch);
                    refresh();
            }
}
```

2.2. Life

This program fragment models the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This code, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
static char sccsid[] = "@(#)life.c                6.1 (Berkeley) 4/23/86";
#endif not lint

# include        <curses.h>
# include        <signal.h>

/*
 *        Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

typedef struct lst_st {                          /* linked list element */
        int             y, x;                    /* (y, x) position of piece */
        struct lst_st   *next, *last;            /* doubly linked */
} LIST;

LIST    *Head;                          /* head of linked list */

int     die();

main(ac, av)
int     ac;
char    *av[];
{
        evalargs(ac, av);                        /* evaluate arguments */

        initscr();                               /* initialize screen package */
        signal(SIGINT, die);                     /* set to restore tty stats */
        cbreak();                                /* set for char-by-char */
        noecho();                                /*
        nonl();                                  /* for optimization */

        getstart();                              /* get starting position */
        for (;;) {
                prboard();                       /* print out current board */
                update();                        /* update board position */
        }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
die()
{
```

```
        signal(SIGINT, SIG_IGN);                    /* ignore rubouts */
            mvcur(0, COLS - 1, LINES - 1, 0);       /* go to bottom of screen */
            endwin();                               /* set terminal to good state
            exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstart()
{
            reg char            c;
            reg int             x, y;
            auto char           buf[100];

            box(stdscr, '|', '_');                  /* box in the screen */
            move(1, 1);                             /* move to upper left corner */

            for (;;) {
                        refresh();                  /* print current position */
                        if ((c = getch()) == 'q')
                                    break;
                        switch (c) {
                          case 'u':
                          case 'i':
                          case 'o':
                          case 'j':
                          case 'l':
                          case 'm':
                          case ',':
                          case '.':
                                    adjustyx(c);
                                    break;
                          case 'f':
                                    mvaddstr(0, 0, "File name: ");
                                    getstr(buf);
                                    readfile(buf);
                                    break;
                          case 'x':
                                    addch('X');
                                    break;
                          case ' ':
                                    addch(' ');
                                    break;
                        }
            }
```

```
        if (Head != NULL)                                   /* start new list */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
        for (y = 1; y < LINES - 1; y++)
                for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                                addlist(y, x);
                }
}

/*
 * Print out the current board position from the linked list
 */
prboard() {

        reg LIST        *hp;

        erase();                                    /* clear out last position */
        box(stdscr, '|', '_');                      /* box in the screen */

        /*
         * go through the list adding each piece to the newly
         * blank board
         */
        for (hp = Head; hp; hp = hp->next)
                mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

3. Motion optimisation

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
```

```
static char sccsid[] = "@(#)twinkle2.c          6.1 (Berkeley) 4/24/86";
#endif not lint

extern int          _putchar();

main()
{
        reg char            *sp;

        srand(getpid());                        /* initialize random sequence */

        if (isatty(0)) {
            gettmode();
            if ((sp = getenv("TERM")) != NULL)
                    setterm(sp);
                signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", _tty_ch);
                exit(1);
        }
        _puts(TI);
        _puts(VS);

        noecho();
        nonl();
        tputs(CL, NLINES, _putchar);
        for (;;) {
                makeboard();                    /* make the board setup */
                puton('*');                     /* put on *'s */
                puton(' ');                     /* cover up with  's */
        }
}

puton(ch)
char    ch;
{
        reg LOCS            *lp;
        reg int             r;
        reg LOCS            *end;
        LOCS                temp;
        static int          lasty, lastx;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++)
```

```
                /* prevent scrolling */
        if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                mvcur(lasty, lastx, lp->y, lp->x);
                putchar(ch);
                lasty = lp->y;
                if ((lastx = lp->x + 1) >= NCOLS)
                        if (AM) {
                                lastx = 0;
                                lasty++;
                        }
                        else
                                lastx = NCOLS - 1;
        }
}
```

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles 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 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 grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

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 describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (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 user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C Ritchie Kernighan Language Prentice and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

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 enclosed in 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.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would 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 also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
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 little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible 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, provided as part of the input specifications, permits 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 represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; 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 theory underlying Yacc has been described elsewhere. Aho Johnson Surveys LR Parsing Aho Johnson Ullman Ambiguous Grammars Aho Ullman Principles Compiler Design Yacc has been extensively used in numerous practical applications, including *lint*, Johnson Lint the Portable C Compiler, Johnson Portable Compiler Theory and a system for typesetting mathematics. Kernighan Cherry typesetting system CACM

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. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often 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 percent "%" 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 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. 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. 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'   newline
'\r'   return
'\''   single quote "'"
'\\'   backslash "\"
'\t'   tab
'\b'   backspace
'\f'   form feed
'\xxx'      "xxx" in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) 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 easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser 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 function returns to its caller after the endmarker is seen; 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. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified 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.

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 value. For example, an action that 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 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, consider the rule

```
expr :   '(' expr ')'   ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr :   '(' expr ')'       { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A   :   B   ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A   :   B
            { $$ = 1; }
        C
            {  x = $2;   y = $3; }
    ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT    :    /* empty */
             { $$ = 1; }
    ;

A   :    B $ACT C
             { x = $2;  y = $3; }
    ;
```

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 to build and maintain the tree structure desired. For example, suppose there is 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 the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr :    expr '+' expr
             { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number 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 file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* 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 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token 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 token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk. Lesk Lex These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF    shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.     reduce 18
```

refers to *grammar rule* 18, while the action

```
IF    shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A   :   x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A     goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then

behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme     :     sound place
          ;
sound     :     DING DONG
          ;
place     :     DELL
          ;
```

When Yacc is invoked with the –v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
	$accept : _rhyme $end

	DING shift 3
	. error

	rhyme goto 1
	sound goto 2

state 1
	$accept : rhyme_$end

	$end accept
	. error

state 2
	rhyme : sound_place

	DELL shift 5
	. error

	place goto 4

state 3
	sound : DING_DONG

	DONG shift 6
	. error

state 4
	rhyme : sound place_ (1)

	. reduce 1

state 5
	place : DELL_ (3)

	. reduce 3

state 6
	sound : DING DONG_ (2)

	. reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token,

DONG, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound    goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that 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 the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *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 that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr*(the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. 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 input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; 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. This is called 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*. Note that there are never any "Shift/shift" conflicts.

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 describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

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 reduce/reduce conflicts should be avoided whenever possible.

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. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts 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 parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, 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
      ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

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 having 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, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced 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 – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

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.

The conflict messages of Yacc are best understood by examining the verbose (**-v**) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat : IF ( cond ) stat_          (18)
        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 ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. 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*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will 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, one of the theoretical references Aho Johnson Surveys Parsing Aho Johnson Ullman Deterministic Ambiguous Aho Ullman Principles Design might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing 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 that 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 that 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, that 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. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes 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 following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr :    expr '+' expr
     |    expr '-' expr
     |    expr '*' expr
     |    expr '/' expr
     |    '-' expr    %prec '*'
     |    NAME
     ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

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 that 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. 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 has no precedence and associativity, 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.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may 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. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: 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 generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves 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 pops its stack until it enters a state where the token "error" is legal. It then

behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, 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 message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat  :     error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat  :     error  ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input:      error  '\n'  {  printf( "Reenter last line: " );  }  input
                    {     $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input:      error  '\n'
                    {     yyerrok;
                          printf( "Reenter last line: " );  }
            input
                    {     $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that 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; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat :    error
              {    resynch();
                   yyerrok ;
                   yyclearin ;  }
     ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function 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 and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **–ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
     return( yyparse() );
     }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
     fprintf( stderr, "%s\n", s );
     }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, 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.

9: 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.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

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. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three 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.

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

```
seq    :     item
       |     seq item
       ;
```

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.

With right recursive rules, such as

```
seq    :     item
       |     item seq
       ;
```

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.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq   :     /* empty */
      |     seq  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. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
      int dflag;
%}
  ... other declarations ...

%%

prog :      decls  stats
      ;

decls :     /* empty */
                  {     dflag = 1; }
      |     decls  declaration
      ;

stats :     /* empty */
                  {     dflag = 0; }
      |     stats  statement
      ;

    ... other rules ...
```

The flag *dflag* is now 0 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. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

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; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent :    adj noun verb adj noun
              { look at the sentence . . . }
     ;

adj  :    THE       {    $$ = THE; }
     |    YOUNG     {    $$ = YOUNG; }
     . . .
     ;

noun :    DOG
              {    $$ = DOG; }
     |    CRONE
              {    if( $0 == YOUNG ){
                       printf( "what?\n" );
                       }
                   $$ = CRONE;
                   }
     ;
     . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint* Johnson Lint

Checker 1273 will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can **not** easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **-d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule :  aaa { $<intval>$ = 3; } bbb
            {   fun( $<intval>2, $<other>0 ); }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kerighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys*, vol. 6, no. 2, pp.99-124, June 1974.
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.*, vol. 18, no. 8, pp. 441-452, August 1975.
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65, 1978. Reprinted as PS1:9 in *UNIX Programmer's Manual*, Usenix Association, (1986).
6. S. C. Johnson, "A portable Compiler: Theory and Practice," *Proc. 5th ACM Symph. on Principles of Programming Languages*, pp. 97-104, January 1978.
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.*, vol. 18, pp. 151-157, Bell Laboratories, Murray Hill, New Jersey, March 1975. Reprinted as USD:26 in *UNIX User's Manual*, Usenix Association, (1986).
8. M. E. Lesk, "Lex - A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual*, Usenix Association, (1986).

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 at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that 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 how precedences and ambiguities are used, and demonstrating simple error recovery. 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; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS     /* supplies precedence for unary minus */

%%     /* beginning of rules section */

list    :       /* 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
                {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                {       $$ = 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 through 9 */
          /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (but not literal) followed by colon */
%token  NUMBER                  /* [0-9]+ */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
```

```
        |   TYPE
            ;

tag     :       /* empty: union tag is optional */
            |   '<' IDENTIFIER '>'
            ;

nlist   :       nmno
            |   nlist nmno
            |   nlist ',' nmno
            ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
            |   IDENTIFIER NUMBER       /* NOTE: illegal with %type */
            ;

            /* rules section */

rules   :       C_IDENTIFIER rbody prec
            |   rules rule
            ;

rule    :       C_IDENTIFIER rbody prec
            |   '|' rbody prec
            ;

rbody   :       /* empty */
            |   rbody IDENTIFIER
            |   rbody act
            ;

act     :       '{' { Copy action, translate $$, etc. } '}'
            ;

prec    :       /* empty */
            |   PREC IDENTIFIER
            |   PREC IDENTIFIER act
            |   prec ';'
            ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, –, *, /, unary –, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 – 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
     double lo, hi;
     } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union  {
     int ival;
     double dval;
     INTERVAL vval;
     }

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST      /* floating point constant */

%type <dval> dexp      /* expression */

%type <vval> vexp      /* interval expression */

     /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS      /* precedence for unary minus */

%%

lines :     /* empty */
     |      lines line
     ;

line :      dexp '\n'
                {     printf( "%15.8f\n", $1 ); }
     |      vexp '\n'
                {     printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
     |      DREG '=' dexp '\n'
                {     dreg[$1] = $3; }
     |      VREG '=' vexp '\n'
```

```
                { vreg[$1] = $3; }
        | error '\n'
                { yyerrok; }
        ;

dexp :  CONST
        | DREG
                { $$ = dreg[$1]; }
        | dexp '+' dexp
                { $$ = $1 + $3; }
        | dexp '-' dexp
                { $$ = $1 - $3; }
        | dexp '*' dexp
                { $$ = $1 * $3; }
        | dexp '/' dexp
                { $$ = $1 / $3; }
        | '-' dexp %prec UMINUS
                { $$ = - $2; }
        | '(' dexp ')'
                { $$ = $2; }
        ;

vexp :  dexp
                { $$.hi = $$.lo = $1; }
        | '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        | VREG
                { $$ = vreg[$1]; }
        | vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        | dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        | vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        | dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi; }
        | vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        | dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        | vexp '/' vexp
                { if( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
    |   dexp '/' vexp
            {   if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1, $1, $3 ); }
    |   '-' vexp  %prec UMINUS
            {   $$.hi = -$2.lo;   $$.lo = -$2.hi;   }
    |   '(' vexp ')'
            {   $$ = $2; }
    ;

%%

# define BSZ 50      /* buffer size for floating point numbers */

    /* lexical analysis */

yylex(){
    register c;

    while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

    if( isupper( c ) ){
        yylval.ival = c - 'A';
        return( VREG );
        }
    if( islower( c ) ){
        yylval.ival = c - 'a';
        return( DREG );
        }

    if( isdigit( c ) || c=='.' ){
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

            *cp = c;
            if( isdigit( c ) ) continue;
            if( c == '.' ){
                if( dot++ || exp ) return( '.' );   /* will cause syntax error */
                continue;
                }

            if( c == 'e' ){
                if( exp++ ) return( 'e' );   /* will cause syntax error */
                continue;
                }

            /* end of number */
            break;
            }
        *cp = '\0';
        if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
        else ungetc( c, stdin );  /* push back last char read */
        yylval.dval = atof( buf );
        return( CONST );
        }
    return( c );
    }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
    /* returns the smallest interval containing a, b, c, and d */
    /* used by *, / routines */
    INTERVAL v;

    if( a>b ) { v.hi = a;  v.lo = b; }
    else { v.hi = b;  v.lo = a; }

    if( c>d ) {
        if( c>v.hi ) v.hi = c;
        if( d<v.lo ) v.lo = d;
        }
    else {
        if( d>v.hi ) v.hi = d;
        if( c<v.lo ) v.lo = c;
        }
    return( v );
    }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
    }

dcheck( v ) INTERVAL v; {
    if( v.hi >= 0. && v.lo <= 0. ){
        printf( "divisor interval contains 0.\n" );
        return( 1 );
        }
    return( 0 );
    }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
    }
```

Appendix D: 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 also be delimited by double 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. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.
6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Part 5
Miscellaneous Material

The Answer to All Man's Problems

Tom Christiansen

CONVEX Computer Corporation
POB 833851
3000 Waterview Parkway
Richardson, TX 75083-3851

{uunet,uiucdcs,sun}!convex!tchrist
tchrist@convex.com

The UNIX on-line manual system was designed many years ago to suit the needs of systems at the time, but despite the growth in complexity of typical systems and the need for more sophisticated software, few modifications have been made to it since then. This paper presents the results of a complete rewrite of the man system. The three principal goals were to effect substantial gains in functionality, extensibility, and speed. The secondary goal was to rewrite a basic UNIX utility in the perl programming language to observe how perl affected development time, execution time, and design decisions.

Extensions to the original man system include storing the whatis database in DBM format for quicker access, intelligent handling of entries with multiple names (via .so inclusion, links, or the NAME section), embedded tbl and eqn directives, multiple man trees, extensible section naming possibilities, user-definable section and sub-section search ordering, an indexing mechanism for long man pages, typesetting of man pages, text-previewer support for bit mapped displays, automatic validity checks on the SEE ALSO sections, support for compressed man pages to conserve disk usage, per-tree man macro definitions, and support for man pages for multiple architectures or software versions from the same host.

1. Introduction

The UNIX on-line manual system was designed many years ago to suit the needs of the systems at the time. Since then, despite the growth in complexity of typical systems and the need for more sophisticated software to support them, few modifications of major significance have been made to the program. This paper describes problems inherent in earlier versions of the *man* program, proposes solutions to these problems, and outlines one implementation of these solutions.

2. The Problem

2.1. The Monolithic Approach

One of the most serious problems with the *man* program up to and including the BSD4.2 release was that all man pages on the entire system were expected to reside under a common directory, */usr/man*. There was no notion of separate sets of man pages installed on the same machine in different subdirectories. At large installations, situations commonly arise in which this functionality is desirable. A site may wish to keep vendor-supplied man pages separate from man pages that were developed locally or acquired from some third party. An individual or group may wish to maintain their own set of man pages. Multiple versions of the same software package might be simultaneously installed on the same machine. A heterogeneous environment may want to be able to view man pages for all available architectures from any machine. Given the

requirement that all man pages live in the same directory, these scenarios are difficult to impossible to support.

The *man* program distributed in the BSD4.3 release included the concept of a MANPATH, a colon-delimited list of complete man trees taken either from the user's environment or supplied on the command line. While this was a vast improvement over the previous monolithic approach, several significant problems remained. For one thing, the program still had to use the *access*(2) system call on all possible paths to find out where the man page for a particular topic existed. When the user has a MANPATH containing multiple components, the time needed for the *man* program to locate a man page is often noticeable, particularly when the target man page does not exist.

2.2. Hard-coded Section Names

Another problem with the *man* program unresolved by the BSD4.3 release was that all possible sections in which a man page could reside were hard-coded into the program. This means that while a **manp** section would be recognized, a **manq** directory would not be, and while a **man3f** directory would be recognized, a **man3x11** directory would not be.

Likewise, the possible subsections for a man page were also embedded in the source code, so a man page named something like */usr/man/man3/XmLabel.3x11* would not be found because **3x11** was not in the hard-coded list of viable subsections. Some systems install all man pages stripped of subsection components in the file name. This situation is less than optimal because it proves useful to be able to supply both a *getc*(3f) and a *getc*(3s). Distinguishing between subsections is particularly convenient with the "intro" man pages; a vendor could supply *intro*(3) *intro*(3a), *intro*(3c), *intro*(3f), *intro*(3m), *intro*(3n), *intro*(3r), *intro*(3s), and *intro*(3x) as introductory man pages for the various libraries. However, the task of running *access*(2) on all possible subsections is slow and tedious, requiring recompilation whenever a new subsection is invented.

3. References in the Filesystem

The existing man system had no elegant way to handle man pages containing more than one entry. For example, *string*(3) contains references to *strcat*(3), *strcpy*(3), amongst others. Because the *man* program looks for entries only in the file system, these extra references must be represented as files that reference the base man page. The most common practice is to have a file consisting of a single line telling *troff* to source the other man page. This file would read something like:

```
.so man3/string.3
```

Occasionally, extra references are created with a link in the file system (either a hard link or a symbolic one). Except when using hard links, this method wastes disk blocks and inodes. In any case, the directory gains more entries, slowing down accesses to files in those directories. Logic must be built into the *man* program to detect these extra references. If not, when man pages are reformatted into their cat directories, separate formatted man pages are stored on disk, wasting substantial amounts of disk space on duplicate information. On systems with numerous man pages, the directories can grow so large that all man pages for a given section cannot be listed on the command line at one time because of kernel restrictions on the total length of the arguments to *exec*(2). Because of the need to store reference information in the file system, the problem is only made worse. This often happens in section 3 after the man pages for the X library have been installed, but can occur in other sections as well.

The *makewhatis*(8) program is a Bourne shell script that generates the */usr/lib/whatis* index, and is used by *apropos*(1) and *whatis*(1) to provide one-line summaries of man pages. These programs are part of the *man* system and are often links to each other and sometimes to *man* itself. If any of the man subdirectories contain more files than the shell can successfully

expand on the command line, the *makewhatis* script fails and no index is generated. When this occurs, *whatis* and *apropos* stop working. The *catman*(8) program, used to pre-format raw man pages, suffers from the same problem.

Of course, *makewhatis* wasn't working all that well, anyway. It was a wrapper around many calls to little programs that each did a small piece of the work, making it run slowly. It, too, had a hard-coded pathname for where man pages resided on disk and which sections were permitted. *Makewhatis* didn't always extract the proper information from the man page's NAME section. When it did, this information was sometimes garbled due to embedded *troff* formatting information. But even garbled information was better than none at all. Even so, these programs left some things to be desired. *Apropos* didn't understand regular expression searches, and both it and *whatis* preferred to do their own lookups using basic, unoptimized C functions like *index*(3) rather than using a general-purpose optimized string search program like *egrep*(1).

4. The Solution

4.1. A Real Database

The problem in all these cases appeared to be that the filesystem was being used as a database, and that this paradigm did not hold up well to expansion. Therefore the solution was to move this information into a database for more rapid access. Using this database, *man* and *whatis* need no longer call *access*(2) to test all possible locations for the desired man page. To solve the other problems, *makewhatis*(8) would be recoded so it didn't rely on the shell for looking at directories.

4.2. Coding in Perl

When the project was first contemplated, the perl programming language by Larry Wall was rapidly gaining popularity as an alternative to C for tasks that were either too slow when written as shell scripts, or simply exceeded the shells' somewhat limited capabilities. Since perl was optimized for parsing text, had convenient *dbm*(3x) support built in to it, and the task really didn't seem complex enough to merit a full-blown treatment in C or C++, perl was selected as the language of choice. Having all code written in perl would also help support heterogeneous environments because the resulting scripts could be copied and run on any hardware or software platform supporting perl. No recompilation would be required.

Some concern existed about choosing an interpreted language when one of the issues to address was that of speed. It was decided to do the prototype in perl and, if necessary, translate this into C should performance prove unacceptable.

The first task was to recode *makewhatis*(8) to generate the new *whatis* database using *dbm*. The *directory*(3) routines were used rather than shell globbing to circumvent the problem of large directories breaking shell wildcard expansions. Perl proved to be an appropriate choice for this type of text processing (see Figure 1).

4.3. Database Format

The database entries themselves are conveniently accessed as arrays from perl. To save space and accommodate man pages with multiple references, two kinds of database entries exist: direct and indirect. Indirect entries are simply references to direct entries. For example, indirect entries for *getc*(3s), *getchar*(3s), *fgetc*(3s), and *getw*(3s) all point to the real entry, which is *getc*(3s). Indirect entries are created for multiple entries in the NAME section, for symbolic and hard links, and for **.so** references. Using the NAME section is the preferred method; the others are supported for backwards compatibility.

```
s/\\f([PBIR]|\(..)//g;     # kill font changes
s/\\s[+-]?\d+//g;          # kill point changes
s/\\&//g;                  # and \&
s/\\\((ru|ul)/_/g;         # xlate to '_'
s/\\\((mi|hy|em)/-/g;      # xlate to '-'
s/\\\*\(..//g &&           # no troff strings
  print STDERR "trimmed troff string macro in NAME section of $FILE\n";
s/\\//g;                   # kill all remaining backslashes
s/^\.\\"\s*//;             # kill comments
if (!/\s+-+\s+/) {
  #  ^ otherwise L-devices would be L
  print STDERR "$FILE: no separated dash in $_\n";
  $needcmdlist = 1;        # forgive their braindamage
  s/.*-//;
  $desc = $_;
} else {
  ($cmdlist, $desc) = ( $`, $' );
  $cmdlist =~ s/^\s+//;
}
```

Figure 1 — *makewhatis* excerpt #1

Assuming that the WHATIS array has been bound to the appropriate *dbm* file, storing indirect entries is trivial:

```
$WHATIS{'fgetc'} = 'getc.3s';
```

When a program encounters an indirect entry, such as for *fgetc*, it must make another lookup based on the return value of first lookup (stripped of its trailing extension) until it finds a direct entry. The trailing extension is kept so that an indirect reference to *gtty*(3c) doesn't accidentally pull out *stty*(1) when it really wanted *stty*(3c).

The format of a direct entry is more complicated, because it needs to encode the description to be used by *whatis*(1) as well as the section and subsection information. It can be distinguished from an indirect entry because it contains four fields delimited by control-A's (ASCII 001), which are themselves prohibited from being in any of the fields. The fields are as follows:

1. List of references that point to this man page; this is usually everything to the left of the hyphen in the NAME section.
2. Relative pathname of the file the man page is kept in; this is stored for the indirect entries.
3. Trailing component of the directory in which the man page can be found, such as **3** for **man3**.
4. Description of the man page for use by the *whatis* and *apropos* programs; basically everything to the right of the hyphen in the NAME section.

At first glance, the third field would seem redundant. It would appear that you could derive it from the character after the dot in the second field. However, to support arbitrary subdirectories like **man3f** or **man3x11**, you must also know the name of the directory so you don't look in **man3** instead. Additionally, a long-standing tradition exists of using the **mano** section to store old man pages from arbitrary sections. Furthermore, man pages are sometimes installed in the wrong section. To support these scenarios, restrictions regarding the format of filenames used for man pages were relaxed in *man*, *makewhatis*, and *catman*, but warnings would be issued

by *makewhatis* for man pages installed in directories that don't have the same suffix as the man pages.

4.4. Multiple References to the Same Topic

A problem arises from the fact that the same topic may exist in more than one section of the manual. When a lookup is performed on a topic, you want to retrieve all possible man page locations for that topic. The *whatis* program wants to display them all to the user, while the *man* program will either show all the man pages (if the **-a** flag is given) or sort what it has retrieved according to a particular section and subsection precedence, by default showing entries from section 1 before those from section 2, and so forth. Therefore, each lookup may actually return a list of direct and indirect lookups. This list is delimited by control-B's (ASCII 002), which are stripped from the data fields, should they somehow contain any. The code for storing a direct entry in the *whatis* database is featured in Figure 2.

```
sub store_direct {
  local($cmd, $list, $page, $section, $desc) = @_; # args
  local($datum);

  $datum = join("\001", $list, $page, $section, $desc);

  if (defined $WHATIS{$cmd}) {
    if (length($WHATIS{$cmd}) + length($datum) + 1 > $MAXDATUM) {
      print STDERR "can't store $page -- would break DBM\n";
      return;
    }
    $WHATIS{$cmd} .= "\002"; # append separator
  }
  $WHATIS{$cmd} .= $datum; # append entry
}
```

Figure 2 — *makewhatis* excerpt #2

Notice the check of the new datum's length against the value of MAXDATUM. This is because of the inherent limitations in the implementation of the *dbm*(3x) routines. This is 1k for *dbm* and 4k for *ndbm*. This restriction will be relaxed if a *dbm*-compatible set of routines is written without these size limitations. The GNU *gdbm* routines hold promise, but they were released after the writing of these programs and haven't been investigated yet. In practice, these limits are seldom if ever reached, especially when *ndbm* is used.

5. Other Problems, Other Solutions

The rewrite of *makewhatis*, *catman*, and *man* to understand multiple man trees and to use a database for topic-to-pathname mapping did much to alleviate the most important problems in the existing man system, but several minor problems remained. Since this was a complete rewrite of the entire system, it seemed an appropriate time to address these as well.

5.1. Indexing Long Pages

Several of the most frequently consulted man pages on the system have grown beyond the scope of a quick reference guide, instead filling the function of a detailed user manual. Man pages of this sort include those for shells, window managers, general purpose utilities such as awk and perl, and the X11 man pages. Although these man pages are internally organized into sections and subsections that are easily visible on a hard-copy printout, the on-line man system could not recognize these internal sections. Instead, the user was forced to search through pages of output looking for the section of the man page containing the desired information.

To alleviate this time-consuming tedium, the man program was taught to parse the *nroff* source for man pages in order to build up an index of these sections and present them to the user on demand. See Figure 3 for an excerpt from the *ksh*(1) index page, displayable via the new **–i** switch.

```
Idx  Subsections in ksh.1              Lines
1    NAME                              3
2    SYNOPSIS                          22
3    DESCRIPTION                       15
4    Definitions.                      43
5    Commands.                         338
6    Comments.                         6
7    Aliasing.                         107
8    Tilde Substitution.               47
9    Command Substitution.             28
10   Process Substitution.             49
11   Parameter Substitution.           645
12   Blank Interpretation.             15
13   File Name Generation.             87
```

Figure 3 — *ksh* index excerpt

The */usr/man/idx*/* directories serve the same function for saved indices as */usr/man/cat*/* directories do for saved formatted man pages. These are regenerated as needed according the the same criteria used to regenerate the cat pages. They can be used to index into a given man page or to list a man page's subsections. To begin at a given subsection, the user appends the desired subsection to the name of the man page on the command line, using a forward slash as a delimiter. Alternatively, the user can just supply a trailing slash on the man page name, in which case they are presented with the index listing like the one the **–i** switch provides, then prompted for the section in which they are interested. A double slash indicates an arbitrary regular expression, not a section name. This is merely a short-hand notation for first running man and then typing

`/expr` from within the user's pager. See Figure 4 for example usages of the indexing features.

```
man -i ksh     # show sections
man ksh/       # show sections, prompt for which one

man ksh/tilde
man ksh/8      # equivalent to preceding line

man ksh/file
man ksh/generat # equivalent to preceding line
man ksh/13     # so is this

man ksh//hangup # start at this string
```

Figure 4 — Index Examples

This indexing scheme is implemented by searching the index stored in */usr/man/idx1/ksh.1* if it exists, or generated dynamically otherwise, for the requested subsection. A numeric subsection is easily handled. For strings, a case-insensitive pattern match is first made anchored to the front of the string, then — failing that — anywhere in the section description. This way the user

doesn't need to type the full section title. The *man* program starts up the pager with a leading argument to begin at that section. Both *more*(1) and *less*(1) understand this particular notation. In the first example given above, this would be

```
less '+/^[ \t]*Tilde Substitution' /usr/man/cat1/ksh.1
```

Once again, perl proved useful for coding this algorithm concisely. The subroutine for doing this is given in Figure 5. Given an expression such as "5" or "tilde" or "file" and a pathname of the man page, *man* loads an array of subsection index titles and quickly retrieves the proper header to pass on to the pager. Perl's built-in **grep** routine for selecting from arrays those elements conforming to certain criteria made the coding easy.

```
sub find_index {
   local($expr, $path) = @_; # subroutine args
   local(@matches, @ssindex);
   @ssindex = &load_index($path);

   if ($expr > 0) {        # test for numeric section
      return $ssindex[$expr];
   } else {
      if (@matches = grep (/^$expr/i, @ssindex)) {
         return $matches[0];
      } elsif (@matches = grep (/$expr/i, @ssindex)) {
         return $matches[0];
      } else {
         return '';
      }
   }
}
```

Figure 5 — Locate Subsection by Index

5.2. Conditional Tbl and Eqn Inclusion

Several other relatively minor enhancements were made to the man system in the course of its rewrite. One of these was to include calls to *eqn*(1) and *tbl*(1) where appropriate. For instance, the X11 man pages use *tbl* directives to construct a number of tables. It was not sufficient to supply these extra filters for all man pages. Besides the slight performance degradation this would incur, a more serious problem exists: some systems have man pages that contain embedded .TS and .TE directives; however, the data between them was not *tbl* input, but rather its output. They have already been pre-processed in the unformatted versions. To do so again causes *tbl* to complain bitterly, so heuristics to check for this condition were built in to the function that determines which filters are needed.

To support tables and equations in man pages when viewed on-line, the output must be run through *col*(1) to be legible. Unfortunately, this strips the man pages of any bold font changes, which is undesirable because it is often important to distinguish between bold and italics for clarity. Therefore, before the formatted man page is fed to *col*, all text in bold (between escape sequences) is converted to character-backspace-character combinations. These combinations can be recognized by the user's pager as a character in a bold font, just as underbar-backspace-character is recognized as an italic (or underlined) one. Unfortunately, while *less* does recognize this convention, *more* does not. By storing the formatted versions with all escape-sequences removed, the user's pager can be invoked without a pipe to *ul* or *col* to fix the reverse line

motion directives. This provides the pager with a handle on the pathname of the cat page, allowing users to back up to the start of man pages, even exceptionally long ones, without exiting the *man* program. This would not be feasible if the pager were being fed from a pipe.

5.3. Troffing and Previewing Man Pages

Now that many sites have high-quality laser printers and bit-mapped displays, it seemed desirable for *man* to understand how to direct *troff* output to these. A new option, **-t**, was added to mean that *troff* should be used instead of *nroff*. This way users can easily get pretty-printed versions of their man pages.

For workstation or X-terminal users, *man* will recognize a TROFF environment variable or command line argument to indicate an alternate program to use for typesetting. (This presumes that the program recognizes *troff* options.) This method often produces more legible output than *nroff* would, allows the user to stay in their office, and saves trees as well.

5.4. Section Ordering

The same topic can occur in more than one section of the manual, but not all users on the system want the same default section ordering that *man* uses to sort these possible pages. For instance, C programmers who want to look up the man page for *sleep*(3) or *stty*(3) find that by default, *man* gives them *sleep*(1) and *stty*(1) instead. A FORTRAN programmer may want to see *system*(3f), but instead gets *system*(3). To accommodate these needs, the *man* program will honor a MANSECT environment variable (or a **–S** command line switch) containing a list of section suffixes. If subsection or multi-character section ordering is desired, this string should be colon-delimited. The default ordering is "ln16823457po". A C programmer might set his MANSECT to be "231" instead to access subroutines and system calls before commands of the same name. A FORTRAN programmer might prefer "3f:2:3:1" to get at the FORTRAN versions of subroutines before the standard C versions. Sections absent from the MANSECT have a sorting priority lower than any that are present.

5.5. Compressed Man Pages

Because man pages are ASCII text files, they stand to benefit from being run through the *compress*(1) program. Compressing man pages typically yields disk space savings of around 60%. The start-up time for decompressing the man page when viewing is not enough to be bothersome. However, running *makewhatis* across compressed man pages takes significantly longer than running it over uncompressed ones, so some sites may wish to keep only the formatted pages compressed, not the unformatted ones.

Two different ways of indicating compressed man pages seem to exist today. One is where the man page itself has an attached **.Z** suffix, yielding pathnames like */usr/man/man1/who.1.Z*. The other way is to have the section directory contain the **.Z** suffix and have the files named normally, as in */usr/man/man1.Z/who.1*. Either strategy is supported to ease porting the program to other systems. All programs dealing with man pages have been updated to understand man pages stored in compressed form.

5.6. Automated Consistency Checking

After receiving a half-dozen or so bug reports regarding non-existent man pages referenced in SEE ALSO sections, it became apparent that the only way to verify that all bugs of this nature had really been expurgated would be to automate the process. The *cfman* program verifies that man pages are mutually consistent in their SEE ALSO references. It also reports man pages whose .TH line claims the man page is in a different place than *cfman* found it. *Cfman* can locate man pages that are improperly referenced rather than merely missing. It can be run on an entire man tree, or on individual files as an aid to developers writing new man pages.

The amount of output produced by *cfman* is startling. A portion of the output of a sample run is seen in Figure 6. Some of its complaints are relatively harmless, such as *dbm* being in

```
at.1: cron(8) really in cron(1)
binmail.1: xsend(1) missing
dbadd.1: dbm(3) really in dbm(3x)
ksh.1: exec(2) missing
ksh.1: signal(2) missing
ksh.1: ulimit(2) missing
ksh.1: rand(3) really in rand(3c)
ksh.1: profile(5) missing
ld.1: fc(1) really in fc(1f)
sccstorcs.1: thinks it's in ci(1)
uuencode.1c: atob(n) missing
yppasswd.1: mkpasswd(5) missing
fstream.3: thinks it's in fstream(3c++)
ftpd.8c: syslog(8) missing
nfmail.8: delivermail(8) missing
versatec.8: vpr(1) missing
```

Figure 6 — Sample *cfman* run

section **3x** rather than section **3**, because the *man* program can find entries with the subsection left off. Having inconsistent .TH headers is also harmless, although the printed man pages will have headers that do not reflect their filenames on the disk. However, entries that refer to pages that are truly absent, like *exec*(2) or *delivermail*(8), merit closer attention.

5.7. Multiple Architecture Support

As mentioned in the discussion of the need for a MANPATH, a site may for various reasons wish to maintain several complete sets of man pages on the same machine. Of course, a user could know to specify the full pathname of the alternate tree on the command line or set up their environment appropriately, but this is inconvenient. Instead, it is preferable to specify the machine type on the command line and let the system worry about pathnames. Consider these examples:

```
man vax csh
apropos sun rpc
whatis tahoe man
```

To implement this, when presented with more than one argument, *man* (in any of its three guises) checks to see whether the first non-switch argument is a directory beneath */usr/man*. If so, it automatically adjusts its MANPATH to that subdirectory.

Not all vendors use precisely the same set of *man*(7) macros for formatting their man pages. Furthermore, it's helpful to see in the header of the man page which manual it came from. The *man* program therefore looks for a local *tmac.an* file in the root of the current man tree for alternate macro definitions. If this file exists, it will be used rather than the system defaults for passing to *nroff* or *troff* when reformatting.

6. Performance Analysis

The *man* program is one that is often used on the system, so users are sensitive to any significant degradation in response time. Because it is written in perl (an interpreted language) this was cause for concern. On a CONVEX C2, the C version runs faster when only one element is present in the MANPATH. However, when the MANPATH contains four elements, the C version bogs down considerably because of the large number of *access*(2) calls it must make.

The start-up time on the parsing of the script, now just over 1300 lines long, is around 0.6 seconds. This time can be reduced by dumping the parse tree that perl generates to disk and executing that instead. The expense of this action is disk space, as the current implementation requires that the whole perl interpreter be included in the new executable, not just the parse tree. This method yields performance superior to that of the C version, irrespective of the number of components in the user's MANPATH, except occasionally on the initial run. This is because the program needs to be loaded into memory the first time. If perl itself is installed "sticky" so it is memory resident, start-up time improves considerably. In any case, the total variance (on a CONVEX) is less than two seconds in the worst case (and often under one second), so it was deemed acceptable, particularly considering the additional functionality the perl version offers.

Nothing in the algorithms employed in the *man* program require that it be written in perl; it was just easier this way. It could be rewritten in C using *dbm*(3x) routines, although the development time would probably be much longer.

The *makewhatis* program was originally a conglomeration of man calls to various individual utilities such as *sed, expand, sort,* and others. The perl rewrite runs in less than half the time of the original, and does a much better job. There are two reasons for the speed increase. The first is the cost of the numerous *exec*(2) calls made via the shell script used by the old version of *makewhatis*. The second is that perl is optimized for text processing, which is most of what *makewhatis* is doing.

Total development time was only a few weeks, which was much shorter than originally anticipated. The short development cycle was chiefly attributable to the ease of text processing in perl, the many built-in routines for doing things that in C would have required extensive library development, and, last but not at all least, the omission of the compilation stage in the normal edit-compile-test cycle of development when working with non-interpreted languages.

7. Conclusions

The system described above has been in operation for the last six months on a large local network consisting of three dozen CONVEX machines, a token VAX, quite a few HP workstations and servers, and innumerable Sun workstations, all running different flavors of UNIX. Despite this heterogeneity, the same code runs on all systems without alterations. Few problems have been seen, and those that did arise were quickly fixed in the scripts, which could be immediately redistributed to the network. The principal project goals of improved functionality, extensibility, and execution time were adequately met, and the experience of rewriting a set of standard UNIX utilities in perl was an educational one. Man pages stand a much better chance of being internally consistent with each other. Response from the user and development community has been favorable. They have been relieved by the many bug fixes and pleasantly surprised by the new functionality. The suite of man programs will replace the old man system in the next release of CONVEX utilities.

Tom Christiansen left the University of Wisconsin with an MS-CS in 1987 where he had been a system administrator for 6 years to join CONVEX Computer Corporation in Richardson, Texas. He is a software development engineer in the Internal Tools Group there, designing software tools to streamline software development and systems administration and to improve overall system security.

Timed Installation and Operation Guide

Riccardo Gusella, Stefano Zatti, James M. Bloom

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

Kirk Smith

Engineering Computer Network
Department of Electrical Engineering
Purdue University
West Lafayette, IN 47906

Introduction

The clock synchronization service for the UNIX 4.3BSD operating system is composed of a collection of time daemons (*timed*) running on the machines in a local area network. The algorithms implemented by the service is based on a master-slave scheme. The time daemons communicate with each other using the *Time Synchronization Protocol* (TSP) which is built on the DARPA UDP protocol and described in detail in [4].

A time daemon has a twofold function. First, it supports the synchronization of the clocks of the various hosts in a local area network. Second, it starts (or takes part in) the election that occurs among slave time daemons when, for any reason, the master disappears. The synchronization mechanism and the election procedure employed by the program *timed* are described in other documents [1,2,3]. The next paragraphs are a brief overview of how the time daemon works. This document is mainly concerned with the administrative and technical issues of running *timed* at a particular site.

A *master time daemon* measures the time differences between the clock of the machine on which it is running and those of all other machines. The master computes the *network time* as the average of the times provided by nonfaulty clocks.[1] It then sends to each *slave time daemon* the correction that should be performed on the clock of its machine. This process is repeated periodically. Since the correction is expressed as a time difference rather than an absolute time, transmission delays do not interfere with the accuracy of the synchronization. When a machine comes up and joins the network, it starts a slave time daemon which will ask the master for the correct time and will reset the machine's clock before any user activity can begin. The time daemons are able to maintain a single network time in spite of the drift of clocks away from each other. The present implementation keeps processor clocks synchronized within 20 milliseconds.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089, and by the CSELT Corporation of Italy. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Government, or of CSELT.

[1] A clock is considered to be faulty when its value is more than a small specified interval apart from the majority of the clocks of the other machines [1,2].

To ensure that the service provided is continuous and reliable, it is necessary to implement an election algorithm to elect a new master should the machine running the current master crash, the master terminate (for example, because of a run-time error), or the network be partitioned. Under our algorithm, slaves are able to realize when the master has stopped functioning and to elect a new master from among themselves. It is important to note that, since the failure of the master results only in a gradual divergence of clock values, the election need not occur immediately.

The machines that are gateways between distinct local area networks require particular care. A time daemon on such machines may act as a *submaster*. This artifact depends on the current inability of transmission protocols to broadcast a message on a network other than the one to which the broadcasting machine is connected. The submaster appears as a slave on one network, and as a master on one or more of the other networks to which it is connected.

A submaster classifies each network as one of three types. A *slave network* is a network on which the submaster acts as a slave. There can only be one slave network. A *master network* is a network on which the submaster acts as a master. An *ignored network* is any other network which already has a valid master. The submaster tries periodically to become master on an ignored network, but gives up immediately if a master already exists.

Guidelines

While the synchronization algorithm is quite general, the election one, requiring a broadcast mechanism, puts constraints on the kind of network on which time daemons can run. The time daemon will only work on networks with broadcast capability augmented with point-to-point links. Machines that are only connected to point-to-point, non-broadcast networks may not use the time daemon.

If we exclude submasters, there will normally be, at most, one master time daemon in a local area internetwork. During an election, only one of the slave time daemons will become the new master. However, because of the characteristics of its machine, a slave can be prevented from becoming the master. Therefore, a subset of machines must be designated as potential master time daemons. A master time daemon will require CPU resources proportional to the number of slaves, in general, more than a slave time daemon, so it may be advisable to limit master time daemons to machines with more powerful processors or lighter loads. Also, machines with inaccurate clocks should not be used as masters. This is a purely administrative decision: an organization may well allow all of its machines to run master time daemons.

At the administrative level, a time daemon on a machine with multiple network interfaces, may be told to ignore all but one network or to ignore one network. This is done with the *-n network* and *-i network* options respectively at start-up time. Typically, the time daemon would be instructed to ignore all but the networks belonging to the local administrative control.

There are some limitations to the current implementation of the time daemon. It is expected that these limitations will be removed in future releases. The constant NHOSTS in /usr/src/etc/timed/globals.h limits the maximum number of machines that may be directly controlled by one master time daemon. The current maximum is 29 (NHOSTS – 1). The constant must be changed and the program recompiled if a site wishes to run *timed* on a larger (inter)network.

In addition, there is a *pathological situation* to be avoided at all costs, that might occur when time daemons run on multiply-connected local area networks. In this case, as we have seen, time daemons running on gateway machines will be submasters and they will act on some of those networks as master time daemons. Consider machines A and B that are both gateways between networks X and Y. If time daemons were started on both A and B without constraints, it would be possible for submaster time daemon A to be a slave on network X and the master on network Y, while submaster time daemon B is a slave on network Y and the master on network X. This *loop* of master time daemons will not function properly or guarantee a unique time on both networks, and will cause the submasters to use large amounts of system resources in the

form of network bandwidth and CPU time. In fact, this kind of *loop* can also be generated with more than two master time daemons, when several local area networks are interconnected.

Installation

In order to start the time daemon on a given machine, the following lines should be added to the *local daemons* section in the file */etc/rc.local*:

```
if [ -f /etc/timed ]; then
        /etc/timed flags & echo -n ' timed' >/dev/console
fi
```

In any case, they must appear after the network is configured via ifconfig(8).

Also, the file */etc/services* should contain the following line:

```
timed       525/udp          timeserver
```

The *flags* are:

-n network to consider the named network.

-i network to ignore the named network.

-t to place tracing information in */usr/adm/timed.log*.

-M to allow this time daemon to become a master. A time daemon run without this option will be forced in the state of slave during an election.

Daily Operation

Timedc(8) is used to control the operation of the time daemon. It may be used to:

- measure the differences between machines' clocks,
- find the location where the master *timed* is running,
- cause election timers on several machines to expire at the same time,
- enable or disable tracing of messages received by *timed*.

See the manual page on *timed*(8) and *timedc*(8) for more detailed information.

The *date(1)* command can be used to set the network date. In order to set the time on a single machine, the *-n* flag can be given to date(1).

References

1. R. Gusella and S. Zatti, *TEMPO: A Network Time Controller for Distributed Berkeley UNIX System*, USENIX Summer Conference Proceedings, Salt Lake City, June 1984.

2. R. Gusella and S. Zatti, *Clock Synchronization in a Local Area Network*, University of California, Berkeley, Technical Report, *to appear.*

3. R. Gusella and S. Zatti, *An Election Algorithm for a Distributed Clock Synchronization Program*, University of California, Berkeley, CS Technical Report #275, Dec. 1985.

4. R. Gusella and S. Zatti, *The Berkeley UNIX 4.3BSD Time Synchronization Protocol*, UNIX Programmer's Manual, 4.3 Berkeley Software Distribution, Volume 2c.

The Transitive Property of Insecurity

Tom Christiansen

CONVEX Computer Corporation
POB 833851
3000 Waterview Parkway
Richardson, TX 75083-3851

{uunet,uiucdcs,sun}!convex!tchrist
tchrist@convex.com

Most UNIX system administrators are at least somewhat concerned with the security of their systems, but many don't know what to look for or where to start. Here are some basic guidelines to establish and maintain system security on a UNIX system.

Introduction

Since the Internet Worm of 1988 and the ensuing publicity, users and administrators of UNIX systems have become increasingly aware of the need for security. UNIX was designed to ease cooperation between users. It is this very design principle that makes UNIX systems particularly vulnerable to security breaches. A substantial portion of the security holes on most UNIX systems are caused not by flaws in the logic of system programs, but rather by carelessness or ignorance on the part of system administrators. Not all systems are managed by people with experience in the subtleties of security in UNIX. Nonetheless, a few straight-forward guidelines can be easily enumerated, which if diligently observed, go a long way towards making a system more secure. Roughly speaking, these amount to establishing and then maintaining a secure environment. These guidelines are applicable to nearly any UNIX system, regardless of flavor.

Physical Security

No treatment of computer security can be considered even moderately complete without mentioning physical security. If the system console is accessible without supervision or restriction, then the system can generally be brought down to single-user mode, at which point the intruder has given himself superuser access, his typical goal. Even if the console should be in secure mode, then a power interruption often suffices to bring the system to single-user mode, since often an interrupt on the console while rebooting puts the system in such a state. The system backup tapes must also be stored in a secure place, because otherwise these tapes can be read for protected data by anyone who can acquire them. Worse yet, they could be replaced with the intruder's own tape containing subtly altered utilities. Without physical security, all other measures are rendered useless.

Writable System Files

The most basic thing to check is that critical system files and directories are not world-writable. If they are, then access to the superuser account is easily attained. These files are */*, */etc*, */bin*, */etc/passwd*, */etc/group*, */.profile*, */.login*, */.cshrc*, */usr/lib/crontab*, and */etc/rc**. The system is insecure if any of these contains the "other write" permission (or possibly "group write"; see below.) Even though */etc/passwd* should be safe, if */etc* is not, then anyone can move the password file out of the way and replace it with their own. Likewise if the root directory (/) is writable, then anyone can move */etc* out of the way and replace it with their own, theirs presumably containing a custom-made password file. Any writable directory or executable file in the superuser's PATH compromises security. To check for this, simply become the superuser and issue the command (assuming *csh* is the shell) `ls -l $path | grep ^.......w` and investigate anything it discloses. Furthermore, the directory "." (the current

working directory) should never be placed in the superuser's PATH, since the superuser may run commands from insecure directories like */tmp* where anyone can install any program they want, which might then be executed inadvertently by the superuser.

It turns out that checking for world-writability of these files is not enough to guarantee their security. A group-writable file or directory is also potentially vulnerable. This is because of what could be called the "transitive property of insecurity." A typical scenario is that the system administrator prefers for various reasons to operate under an effective **umask** of 002, thus allowing group write access to files and directories he creates. This applies both to the system administrator's own personal start-up files, such as *.login* or *.profile,* as well as to system directories like */usr/bin* or */usr/local/bin.* Directories like these are often group-writable by **bin** or **staff.** If an intruder can somehow gain those group privileges, then they can also gain superuser privileges.

This problem is particularly insidious on UNIX systems supporting multiple groups because of the potential of mixing groups of high and low privilege in the same account. If the system administrator belongs to groups **staff** and **design**, and any system file or directory is writable by group **staff**, then all the intruder need do is compromise an account in group **design**. He then proceeds to locate and compromise a file or directory writable by group **design** that the system administrator would normally access. This gains him access to group **staff**, which in turn gives him access to the superuser account itself. Carrying the transitive property of insecurity one stem further, if a näive user is a member of groups **design** and **world**, where **world** is a group to which all users pertain, then this means that now any user on the system can become the superuser with just a little bit of effort.

In summary, never leave group write permissions on a system object unless that group is one to which no real users belong. This means that no one, not even a system administrator, should ever belong to system groups like **tty**, **daemon**, or **kmem**.

Once all system files and directories have secure permissions, owners, and groups, make a list of these for later reference. This list should as a bare minimum contain the output of `ls -lgi`, but preferably it will also contain checksum information so that even if a file's byte count and modification date are massaged back to what they were before the breach, if their contents are altered, this will be detected. Now move this list off-line. This is important, because if a system is compromised, then all files on that system are suspect.

Particular care must be given to device files. Files like */dev/mem, /dev/kmem,* all the tty devices (except */dev/tty*) and the raw and block disk devices must be protected from public access if any security is to be maintained. If kernel memory is accessible, anything in the system's memory can be inspected. If the terminal devices are accessible even just for writing, then control characters can be sent to them triggering their answer-back sequences and sending back to the host computer anything the cracker wants. If the system is pre-POSIX (and most are), then if supported the TIOCSTI *ioctl* and the *setpgrp* system call can be used to stuff characters into someone else's input buffer, making it appear as though they were typed in by the user. If that user is the superuser, anything is possible. If the disks are readable, the file system's protection mechanism can be circumvented. Device security is essential.

Setuid Programs

Apart from incorrect file permissions, setuid programs pose the greatest threat to system security. This is because they allow to do in a (theoretically) controlled fashion precisely what an intruder is trying to do: run as the superuser. Although set user id to **root** programs are the most obviously dangerous, the transitive property of insecurity could be applied to any setuid or setgid program, demonstrating that they, too, pose a threat. Setuid programs should be used only when absolutely necessary. If properly installed, a setgid program will usually suffice for the task at hand and will be more secure than a setuid one would be. Always know where all the setuid and setgid programs on your system are, what each one of them is for, and why it is necessary. This is easily checked using the `ncheck -s` command. This list generated by *ncheck* should be taken off-line for the same reasons as the previously described *ls* output should be. Periodically run *diff* on the current list and the previous one to find any changes, and investigate any discrepancies rigorously.

Writing a setuid program is difficult because it overrides the protection mechanism of the system. If this can't be avoided, be exceedingly careful before installing it. Make sure that the effective uid is reset before calling another program, either via *exec*, *popen*, or *system*. Always use full path names when calling one of these functions. Make sure that the environment is stripped of unnecessary variables, and that needed ones are carefully checked or set to something secure; PATH and IFS are particularly sensitive to penetration attempts. Make sure the process's **umask** is appropriately set and that no files are created with write permission for anyone other than their intended user. Close all unneeded open file descriptors. Log any activity that affects the security of the system, preferably using *syslog* with the facility class LOG_AUTH if available. Periodically examine all system log files for questionable activity.

The setuid shell script must be avoided at all costs. On systems that support these, a subtle but significant bug usually exists in the operating system that can be exploited by an intruder. The bug is that there exists a window of vulnerability between the time that the kernel decides a program should be run setuid and when the now-setuid shell opens the script to interpret it. If symbolic links are available, or if the file system on which the setuid script resides has a writable directory anywhere on it, then the intruder can easily arrange to change the file being interpreted by switching the link. In short, never use setuid shell scripts.

Because of the transitive property of insecurity, any program ever run by the superuser, even those that are not themselves setuid, falls under the same set of cautions that setuid program do. This all includes programs directly or indirectly run out of */usr/lib/crontab* and */etc/rc** during the boot process, as well as those run by the superuser during routine maintenance work. Even seemingly harmless programs can be the unwitting mechanism by which the system is compromised if they are run by the superuser. If the editor looks for start-up files in the current working directly irrespective of their ownership, then merely running the editor in a world-writable directory like */tmp* can render the system vulnerable.

Often, system administrators install setuid shells for themselves to avoid always having to type in a password when they wish to become the superuser. The is imprudent in the extreme, because it means that to become superuser, the intruder need only become the system administrator, which is generally far easier to do. Private versions of *su* like this should never be allowed.

Passwords

Choosing a good password is essential to good security. Periodically changing it is also important. Some guidelines for choosing a good password are that you should never choose a word in the dictionary, one shorter than six characters, one containing the user's real name or login id, or one that doesn't contain some non-alphanumeric characters. Require that every account have a password, preferably one conforming to these criteria. As a result of the increasing availability of extremely fast processors, custom decryption chips, and vectorizable DES encryption algorithms, even these measures can prove insufficient. The solution is to use a shadow password scheme in which the encrypted passwords are not available for public perusal. Under such a scheme, the passwords in */etc/passwd* are replaced by asterisks and the real encrypted passwords are stored in a protected file. If your vendor does not support this, strongly advise them to do so.

Networking Hazards

A system attached to a network is inherently less secure than an isolated one. Some problems are difficult to impossible to solve. One such problem is the way in which passwords are transmitted across the net without encryption. An experienced intruder with access to the physical network can pick these packets off the net and use them to reconstruct a user's password. There's little that can be done about this. Just as it is nearly impossible to protect a building with a lock that can't be picked by a locksmith, likewise it is nearly impossible to protect a computer system from a knowledgeable and experienced security expert, licit or otherwise. However, it is possible to install fairly good locks that will keep the general populace honest.

One thing that can be done is to inspect and secure the facilities that permit network access without password validation. The *ftp* program will consult a user's *.netrc* file, and the Berkeley *rlogin*, *rsh*, *rcp*, and *rdist* programs will consult a user's *.rhosts* file as well as the system */etc/hosts.equiv* file. These files should all be readable and writable by no one but their owner.

exists the other direction; this information should be protected. The transitive property of insecurity is once again at work here: if a host anywhere in the chain is compromised, all indirectly connected hosts are also compromised. Even if the user is not the superuser, a mere user account is enough to get a foot in the door and start poking around for holes. Therefore, *.rhosts* files should be used sparingly, should be accessible by no one but there owner, should be owned by the person in whose home directory they reside, and should never contain hosts from outside the immediate administrative domain. The superuser should nearly never have a *.rhosts* file.

The Internet Worm of 1988 quickly spread to thousands of hosts by exploiting *.rhosts* connectivity as well as long-standing bugs in networking programs like *sendmail* and the *finger* daemon. These bugs have since been widely publicized, but despite this publicity, not all vendors have corrected all the problems. Contact your vendor and get a complete list of bug fixes they've applied to plug the holes uncovered by the Internet Worm. Make sure the fixes have really made it into the release on your system, and get patches if they haven't.

NFS

Ease of use and security are usually inversely proportional to each other. Just as the addition of networking to a system increases the usability and decreases the security of the system, the addition of NFS on top of normal networking has the same effect, except more so. Take care not to export any file systems to the whole world: make sure that the file system has an explicit access list in the */etc/exports* file. Make sure that the user and group ids on all involved systems are the same. Do not import a file system with superuser access enabled on the remote system; this is determined by the **nosuid** option on the *mount* command or in */etc/fstab*. Do not import a file system with setuid files enabled.

Even all this is not really good enough for true security. Often a file server or mainframe makes its files systems available to workstations. The problem is that even if remote users can't be the superuser on your exported file systems, they can probably become the superuser on their own desktop system. This means they can become any user they want and gain them access to any file on the host system that's writable by any non-super user. The transitive property of insecurity has shown that this is often enough to compromise a system.

Still worse, even though NFS can impede setuid files from functioning on an imported file system, it doesn't address the issue of devices. A user can create new devices on his workstation as the superuser, making sure their major and minor numbers correspond to the server system's device numbers, then log in to the server and run programs accessing these new devices. The host system will merrily permit access to these newly created devices files referencing protected objects like physical memory or the raw disks. Do not import file systems from anyone you do not trust.

Conclusion

Securing a UNIX system requires active attention by its system administrator. It doesn't require a guru to safeguard against the most common problems. The work may sometimes seems tedious, but it will be well worth the trouble if you ever have an attempted security penetration.

Tom Christiansen left the University of Wisconsin with an MS-CS in 1987 where he had been a system administrator for 6 years to join CONVEX Computer Corporation in Richardson, Texas. He is a software development engineer in the Internal Tools Group there, designing software tools to streamline software development and systems administration and to improve overall system security.

Bibliography

Overview

This annotated bibliography contains third-party documentation that provides technical information useful to users of CONVEX systems and software.

Documentation is divided into these categories:

- Introductory material
- Text editing and document preparation
- Communications
- Supporting tools
- UNIX in general
- Miscellaneous material

Introductory material

Bourne, Philip E. *UNIX for VMS Users*. Digital Press, 1989.
Presentation of fundamental concepts and basic command procedures. Covers the use of high-level languages, programming the operating system, text processing, and networked communications. Appendices cover command and file summaries.

Lamb, Linda. *Learning the vi Editor*. O'Reilly and Associates, 1988.
A complete guide to editing with `vi`. Covers basic editing, moving around quickly, beyond the basics, greater power with `ex`, global search and replacement, customizing `vi` and `ex`, command shortcuts, and quick reference to `vi` and `ex` commands.

Loukides, Mike. *UNIX for FORTRAN Programmers*. O'Reilly and Associates, 1989.
An introduction to UNIX and these UNIX tools: the FORTRAN compiler (f77); UNIX interactive command languages; `vi`; object library management tools (`ar` and `ranlib`); `adb` and `dbx`; `prof`, `gprof`, and time-profiling tools; `make`; and rcs, a source-code management system for large projects.

Todino, G.; Strang, J. *Learning the UNIX Operating System*. O'Reilly and Associates, 1988.
Introduction for new UNIX users. Topics include logging in and out, managing UNIX files and directories, sending and receiving mail, redirecting input/output, pipes and filters, background processing, and customizing your account.

Text editing and document preparation

Dougherty, Dale; O'Reilly, Tim. *UNIX Text Processing*. O'Reilly and Associates, 1988.
Covers basic editing to writing complex `troff` macro packages. Topics include editing with `vi` and `ex`, basic `nroff` and `troff` requests, ms and mm macros, formatting with `tbl`, typesetting equations with `eqn`, drawing pictures with `pic`, shell programming for writers, `awk` programming language, `sed` stream editing, writing `troff` macros, indexing with `troff`, `sed`, and `awk`, and managing book production with `make`.

Gehani, Narain. *Document Formatting and Typesetting on the UNIX System*. Silicon Press, 1986.
Contains a general discussion of formatting and an extensive discussion of the UNIX system document formatting tools. Covers -mm macors, `tbl`, `pic`, `eqn`, and `troff`. Contains templates for preparing a variety of documents and descriptions of the UNIX system typesetting commands.

Communications

Comer, Douglas. *Internetworking with TCP/IP*, Second Edition. Prentice Hall, 1991.
Discusses protocols, addressing, address resolution (ARP), routing, and gateways. Handy for anyone who needs to manage a TCP/IP network, design network applications, or write network programs.

Frey, Donnalyn.; Adams, Rick.!@%: *A Directory of Electronic Mail Addressing & Networks*. O'Reilly and Associates, 1990.
Lists more than 130 networks around the world and, for each one, shows general description, address structure and format, architecture, connections to other networks or sites, facilities available to users, contact name and address, cross-references to other networks, future plans, and the date of last update.

Kochan, Stephen; Wood, Patrick. *UNIX Networking*. Howard W. Sams and Company, 1989.
Comprehensive look at the major aspects of networking in a UNIX system.

Tanenbaum, Andrew. *Computer Networks*, Second Edition. Prentice Hall, 1989.
Contains general networking information, including network architecture, protocols, and applications. Covers TCP/IP, OSI. X.25, NFS, LANs, WANs, FDDI, and Ethernet.

O'Reilly, Tim; Todino, Grace. *Managing UUCP and Usenet*. O'Reilly and Associates, 1989.
Written for system administrators, this book explains how to install and manage UUCP and Usenet software. Covers: how UUCP works, RS-32 cabling, talking with modems, setting up a UUCP link, security considerations, UUCP administrations, introduction to Usenet, installing Netnews, administering Netnews, and working files.

Todino, Grace; Dougherty, Dale. *Using UUCP and Usenet*. O'Reilly and Associates, 1988.
Shows how to use `cu` and `tip` to communicate with both UNIX and non-UNIX systems. For beginners and experienced users. Covers: introduction to UUCP communications, copying files with UUCP, remote command execution, sending mail to distant users, checking on UUCP requests, remote `login` with `cu` or `tip`, extending the UUCP network, and reading and posting news.

The Waite Group. *UNIX Communications*. Howard W. Sams and Company, 1989.
For novice and expert UNIX users, covers mail, networking, and file transfer tools.

UNIX in general

Christian, Kaare. *The C and UNIX Dictionary*. John Wiley and Sons, 1988.
Contains basic computer and computer hardware terminology, terms relating to system maintainence, technical terms relating to programming, and profiles of individuals who have significantly contributed to the development of the UNIX system.

Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California. *UNIX Programmer's Manual Supplementary Documents 1*.1986.
One of the sources for the *ConvexOS Tutorial Papers*. It is programmer-oriented and contains papers on languages and programming tools.

Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California. *UNIX Programmer's Manual Supplementary Documents* 2. 1986.
One of the sources for the *ConvexOS Tutorial Papers*. It is programmer-oriented and contains papers on languages and programming tools.

Foxley, Eric. *UNIX for Super-Users*. Addison-Wesley, 1988.
Describes powering up and powering down the system, creating new `login` names, maintaining file security, monitoring user resource usage, and machine performance considerations.

Kernighan, Brian; Pike, Robert. *The UNIX Programming Environment*. Prentice Hall, 1984.
Covers file systems, using the C shell, filters, shell programming, programming with standard I/O, UNIX system calls, program development, and document preparation.

Lewine, Donald. *POSIX Programmer's Guide*. O'Reilly and Associates, 1991.
Covers introduction to portability, depositing applications, standard file and terminal I/O, files and directories, advanced file operations, processes, obtaining information at run-time, and POSIX and standard C. Includes references sections in library functions, header files, data structures, error codes, porting, changes and additions in standard C, and federal information processing standard 151-1.

Loukides, Mike. *System Performance Tuning*. O'Reilly and Associates, 1990.
Topics covered include real and perceived performance problems, tricks to improve keyboard response, how to manage your system's load, how to survive without a lot of memory, how to configure your I/O system for the best throughput, how to detect on overworked or malfunctioning network, and how to build a more efficient kernel.

Nemeth, Evi; Snyder, Garth; Seebass, Scott. *UNIX System Administration Handbook*. Prentice Hall, 1989.
Covers typical system administration duties, booting and shutting down, superuser privileges, filesystems, controlling processes, adding new users, devices and drivers, configuring the kernel, installing terminals, printing under ATT, printing under BSD, adding a disk, hardware maintenance tips, networking, mail and sendmail, uucp, news, backups and transportable media, accounting, daemons, and periodic processes.

Quarterman, John S., et al. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1988.
Covers the internal structure of the 4.3BSD system and the concepts, data structures, and algorithms used in implementing the system facilities. Kernel discussion includes system process management, memory management, the I/O system, the filesystem, the socket IPC mechanism, and network-protocol implementations.

Sage, Russell G. *Tricks of the UNIX Masters*. Howard W. Sams and Company, 1987.
Covers how UNIX works, task management, security and personal security, UNIX communications, useful tricks (powerful one- and two-line commands)

The Waite Group. *UNIX Papers*. Prentice Hall, 1990.
Collection of papers on a broad range of advanced topics such as security, communications, and standards for the operating system.

Supporting tools

Aho, Alfred; Kernighan, Brian; Wienberger, P. *The awk Programming Language*. Addison-Wesley, 1988.
This book documents `nawk`, the "new" version of `awk`. Users need `nawk` — part of the optional CONVEX Toolbox product — to use this book effectively.

Anderson, Gail; Anderson, Paul. *The UNIX C Shell Field Guide.* Prentice Hall, 1986.
Explains how UNIX works. Focuses on file management and personal security. Covers UNIX communications, reveals powerful one- and two-line commands, and discusses shell programming and debugging.

Curry, Dave. *Using C on the UNIX System.* O'Reilly and Associates, 1988.
Written for intermediate to experienced C programmers who want to be UNIX programmers, this book covers I/O using stdio, manipulating files and directories, device I/O control, getting information about users, telling time and timing things, processing signals, creating processes and executing programs, job control, interprocess communication, and networking (Internet clients, servers).

Darwin, Ian F. *Checking C Programs with lint.* O'Reilly and Associates, 1989.
Contents include an overview of using `lint`, casting and delinting, `lint` comments, command line options, using `lint` with `make`, rolling your own `lint` library, public domain programs, a look inside `lint`, and future directions.

Dougherty, Dale. *sed & awk.* O'Reilly and Associates, 1990.
Covers the original and the new `awk`. Includes how to write a `sed` script, using both basic and advanced commands, how to use `awk`'s built-in functions as well as how to write user-defined functions that are available with `nawk`. Also includes miscellaneous `sed` and `awk` scripts that demonstrate a wide range of scripting styles and techniques.

Gircy, Gintaras R. *Understanding and Using COFF.* O'Reilly and Associates, 1987.
Contents cover the basics of COFF, assembly code relocation process, COFF file headers, relocation structures, the linking process, the COFF system in UNIX, magic numbers, the COFF symbolic debugger, and COFF and shared libraries.

Loukides, Mike. *UNIX for FORTRAN Programmers.* O'Reilly and Associates, 1989
An introduction to UNIX and these UNIX tools: the FORTRAN compiler (f77); UNIX interactive command languages; `vi`; object library management tools (`ar` and `ranlib`); `adb` and `dbx`; `prof`, `gprof`, and time profiling tools, `make`; and `rcs`, a source code management system for large projects.

Mason, Tony; Brown, Doug. *lex and yacc.* O'Reilly and Associates, 1989.
Part one provides a basic understanding of language theory, the function of compilers and interpreters, lex and yacc, and presents tutorial examples. The second part provides detailed information for advanced use of lex and yacc. Appendices cover GNU, Flex, and Bison. This book assumes some knowledge of UNIX and C.

Strang, John. *Programming with curses.* O'Reilly and Associates, 1989.
Curses is a UNIX library of functions for controlling a terminal's screen display. This handbook covers Ken Arnold's original Berkeley implementation of curses, not the System V version. Topics include windows, screens, and images; multiple windows, the WINDOW structure, terminal independence, the curses functions, and sample.

Strang, John; Mui, Linda; O'Reilly, Tim. *termcap & terminfo.* O'Reilly and Associates, 1898.
This book documents hundreds of capabilities of `termcap` and `terminfo`. Also covers: terminal independence, reading `termcap` and `terminfo` entries, capability syntax, initializing the terminal environment, writing `termcap` and `terminfo` entries, converting between `termcap` and `terminfo`, and detailed descriptions of each capability and how it is used.

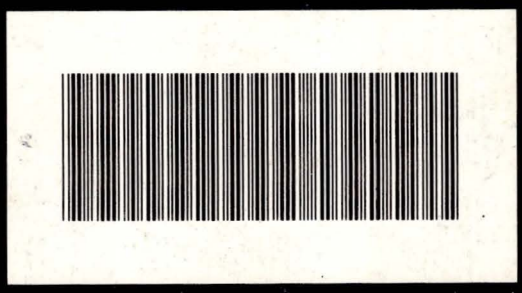
Talbott, Steve. *Managing Projects with make.* O'Reilly and Associates, 1987.
Topics are: writing a simple makefile, shell variables, internal macros, suffix rules, special description targets, maintaining libraries, and invoking `make` recursively.

Miscellaneous material

Dougherty, Dale; O'Reilly, Tim. *DOS Meets UNIX*. O'Reilly and Associates, 1988.
Describes the solutions available for integrating DOS and UNIX. Includes: DOS, UNIX, and departmental computing, strategies to integrate DOS and UNIX, introduction to UNIX, DOS-UNIX networking, the virtual DOS environment, hardware issues, and where to buy products.

Dougherty, Dale. *UNIX in a Nutshell for HyperCard*. O'Reilly and Associates, 1989.
Contains a series of HyperCard stacks for Macintosh users of UNIX systems. Includes: the main stack, introductory stack (demonstrates applicable HyperCard techniques), command stack for both System V and Berkeley commands, text editor stack, shell stack, programming tools stack, freeform searching capabilities, glossary, and topic index.

Kochan, Stephen G.; Wood, Patrick H. *UNIX System Security*. Macmillan, 1990.
Addresses each of the major security issues concerning UNIX users; includes information about administering passwords, auditing security, checking file permissions, data encryption, and setting up a restricted environment.



Order Number
DSW-002

Document Number
710-011130-001