Part Number 800-1113-01
Revision: D of 7th January 1984
For: Sun System Release 1.1

Programming Tools

for the

Sun Workstation

Sun Microsystems, Inc.,
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

Credits and Acknowledgements

The chapters of this manual were originally derived from the work of many people at Bell Laboratories, University of California at Berkeley, and other noble institutions. Their names and the titles of the original works appear here.

Shell Programming
was derived from the papers *An Introduction to the UNIX Shell*, by S. R. Bourne, Bell Laboratories, Murray Hill, New Jersey, and *An Introduction to the C Shell*, by William Joy, University of California at Berkeley.

UNIX Programming
by Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories, Murray Hill, New Jersey.

Lint, a C Program Checker
by S. C. Johnson, Bell Laboratories, Murray Hill, New Jersey.

Make — A Program for Maintaining Computer Programs
by S. I. Feldman, Bell Laboratories, Murray Hill, New Jersey.

DC — An Interactive Desk Calculator
by Robert Morris and Lorinda Cherry, Bell Laboratories, Murray Hill, New Jersey.

BC — An Arbitrary Precision Desk-Calculator Language
by Lorinda Cherry and Robert Morris, Bell Laboratories, Murray Hill, New Jersey.

The M4 Macro Processor
by Brian W. Kernighan and Dennis M. Ritchie, Bell Laboratories, Murray Hill, New Jersey.

Lex — A Lexical Analyzer Generator
by M. E. Lesk and E. Schmidt, Bell Laboratories, Murray Hill, New Jersey.

Yacc — Yet Another Compiler-Compiler
by Stephen C. Johnson, Bell Laboratories, Murray Hill, New Jersey.

Source Code Control System User's Guide
by L. E. Bonanni and C. A. Salemi, Bell Laboratories, Piscataway, New Jersey.

Source Code Control System
by Eric Allman, Formerly of Project Ingres, University of California at Berkeley.

Assembler Reference Manual for the Sun Workstation
started life as an edited version of the MICAL Manual for the Intel 8080, written by Mike Patrick; transformed by James L. Gula and Thomas J. Teixeira, March 1980; revised by Henry McGilton at Unisoft Systems of Berkeley Corporation during March 1982; rewritten by Henry McGilton and Richard Tuck, of Sun Microsystems, during October and November 1982.

PROGRAMMING TOOLS

Contents

This is a list of the major sections in this manual. There is a detailed table-of-contents at the start of each paper.

Table of Contents

Trademarks

Multibus is a trademark of Intel Corporation.

Sun Workstation is a trademark of Sun Microsystems Incorporated.

UNIX is a trademark of Bell Laboratories.

Revision History

Revision	Date	Comments
A	15th July 1983	First release of this Manual.
B	15th August 1983	Second Release of this manual entailed a complete reorganization and some rewriting of the individual articles.
C	1st November 1983	Third Release of this manual entailed minor corrections and updates.
D	7th January 1984	Added chapter on Shell Programming. Added chapter on ADB. Many minor corrections and updates.

Programming the Shells

You can put programs in files called *Shell scripts*, and then call up the Shells to read and execute the commands from these files.

Understand that Shell scripts do not serve the same function as the *make* program. *Make* program 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 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*. Using a *makefile* is superior to maintaining a group of Shell procedures to update these files. Similarly when working on a document, you can create a *makefile*, which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

When you have a file full of Shell commands and you simply type the name of that file as a command, the system looks at the very first line of that file to decide which Shell should run the script:

- If the first line does *not* start with a # (hash sign), the system uses the Bourne Shell to run the script.
- If the first line starts with a # (hash sign) and is *not* followed by a ! (exclamation mark), the system uses the C-Shell to run the script.
- Finally, if the first line of the Shell script starts with a #! combination and is followed immediately by a name, the system looks for a program of that name to run the Shell script.

Part I — Programming the C Shell

This section details C-Shell features useful for writing Shell scripts.

1. Invocation and the Argv Variable

A *csh* command script may be interpreted by saying:

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

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 available through the same mechanisms used to refer to any other Shell variables.

If you make the file *script* executable by changing its permissions with the *chmod* command:

```
tutorial% chmod 755 script
tutorial%
```

and place a Shell comment at the beginning of the Shell script, that is, begin the file with a '#' character, */bin/csh* will then automatically be called to execute *script* when you type:

```
tutorial% script
tutorial%
```

If the file does not begin with a '#', then the standard Shell */bin/sh* executes it. Thus, you can convert your older Shell scripts to use *csh* at your convenience.

2. Variable Substitution

After each input line is broken into words and history substitutions are applied, the input line is parsed into distinct commands. Before each command is executed, the *variable substitution* mechanism is applied on these words. Keyed by the character '$', this substitution replaces the names of variables with their values. Thus, if you place:

```
echo $argv
```

in a command script, the current value of the variable *argv* is 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 available for accessing components and variable attributes. The notation:

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. This 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

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

It is also possible to access the components of a variable that has several values. To get the first component of *argv* or in the example above 'a', use:

```
$argv[1]
```

Similarly to get 'c', use:

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

and to get 'a b', use:

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

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
```

To expand to the process number of the current Shell, use the form:

```
$$
```

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). Use this 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 to write out the prompt 'yes or no?' without a newline and then read the answer into the variable 'a', use:

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

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

Note one minor difference between '$*n*' and '$argv[*n*]'. The form '$argv[*n*]' yields an error if *n* is not in the range '1-$#argv', while '$n' never yields 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. Expressions

To construct interesting Shell scripts, it must be possible to evaluate expressions in the Shell based on the values of variables. In fact, all the arithmetic operations of the C language 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 replaced by a number of single characters. For instance, the expression primitive:

```
-e filename
```

tells 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 }'. This primitive returns true, that is '1', if the command exits 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 '$status' examined in the next command. Since every command sets '$status', 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 user's manual section on the C-Shell.

4. Sample Shell Script

A sample Shell script that uses the Shell expression mechanism and some of its control structure follows:

```
tutorial% 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 uses 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 you 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 that 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 Shell does not accept the following two formats:

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

The Shell does have another form of the *if* statement of the form:

```
if ( expression ) command
```

which can be written

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

Here you escape 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*, for example:

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

else
    commands
endif
```

Use the ':' modifier in Shell scripts, for instance in the modifier ':r' 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:

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

shows how the ':r' modifier strips off the trailing '.bar', and the ':e' modifier leaves only the 'bar'. Other modifiers 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'. See the *csh* pages in the user's manual for a full description of these modifiers.

It is also possible to use the *command substitution* mechanism to perform modifications on strings to then re-enter the Shell's environment. Since calling this mechanism creates a new process each time, it is much more expensive to use than the ':' modification mechanism.#

Finally, note that the character '#' lexically introduces a Shell comment in Shell scripts, but not from the terminal. The Shell discards all subsequent characters on the input line after a '#'. Quote this character using '" or '\' to place it in an argument word.

---

and

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

#Note that the current implementation of the Shell limits the number of ':' modifiers on a '$' substitution to 1. Thus:

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

does not do what one mght expect.

5. Other Control Structures

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

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

and

```
switch ( word )

case str1:
    commands
    breaksw

...

case strn:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

See the user's manual pages on *csh* for details. C programmers should note that *breaksw* exits from a *switch*, while *break* exits a *while* or *foreach* loop. Do not make the common mistake in *csh* scripts of using *break* instead of *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, that is:

```
loop:
    commands
    goto loop
```

6. Supplying Input to Commands

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

Thus use 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:

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

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, that is quoted, prevents the Shell from performing variable substitution on the intervening lines. In general, the Shell uses '<<' to terminate the text to be given to the command. If any part of the word following the '<<' is quoted, these substitutions are not performed. In this case, since you used the form '1,$' in your editor script, you needed to insure that this '$' was not variable substituted. You can also insure this by preceding the '$' here with a '\', for instance:

```
1,\$s/↑[ ]*//
```

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

7. Catching Interrupts with 'onintr'

If your Shell script creates temporary files, you may wish to catch the Shell script interruptions so you can clean up these files. You can then use *onintr* as follows:

```
onintr label
```

where *label* is a label in your program. If the Shell receives an interrupt, it does a 'goto *label*', and you 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 you wish to exit with a non-zero status, do the following:

```
exit(1)
```

that is, to exit with status '1'.

Briefly, there are other Shell features that are useful for writing Shell procedures. You can use the *verbose* and *echo* options and the related –v and –x command line options to help trace the actions of the Shell. The –n option causes the Shell only to read commands and not to execute them.

Also note that *csh* only executes Shell scripts that begin with the character '#', that is, Shell scripts that begin with a comment (assuming that another Shell was not specified via the ! mechanism). 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 '"' that allows only some of the expansion mechanisms to occur on the quoted string and makes this string into a single word as '' does.

8. Other C-Shell Features

This section describes other less commonly used C-Shell features.

8.1. Loops at the Terminal and Variables as Vectors

The *foreach* control structure aids 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, you can say:

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

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

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

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

Variables that contain lists of filenames or other words are very useful with loops. You can, for example, do:

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

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

The Shell converts the output of a command within "`" characters 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. Use a modifier ':x' later to expand each component of the variable into another variable, splitting it into separate words at embedded blanks and tabs.

8.2. Command Substitution

A command enclosed in "`" characters is replaced, just before filenames are expanded, by the output from that command. Thus, to save the current directory in the variable *pwd*, say:

```
set pwd=`pwd`
```

Or to run the *ex* editor, say:

```
ex `grep -l TRACE *.c`
```

This uses those files whose names end in '.c', which have the string 'TRACE' in them as arguments.*

In particular circumstances, you may need to know the exact nature and order of different substitutions that the Shell performs and the exact meaning of certain combinations of quotations. Moreover, the Shell has a number of command line option flags used mostly in writing UNIX programs and debugging Shell scripts. See the user's manual section on *csh* and *sh* for details.

9. Special Characters

The following table lists the special *csh* and UNIX system characters. A number of these characters also have special meaning in expressions. See the *csh* manual section for a complete list.

Syntactic Metacharacters

;	separates commands to be executed sequentially
|	separates commands in a pipeline
( )	brackets expressions and variable values
&	follows commands to be executed without waiting for completion

Filename Metacharacters

/	separates components of a file's pathname
?	expansion character matching any single character
*	expansion character matching any sequence of characters
[ ]	expansion sequence matching any single character from a set
~	used at the beginning of a filename to indicate home directories
{ }	used to specify groups of arguments with common parts

Quotation Metacharacters

\	prevents meta-meaning of following single character
'	prevents meta-meaning of a group of characters
"	like ', but allows variable and command expansion

Input/output Metacharacters

<	indicates redirected input
>	indicates redirected output

Expansion/substitution Metacharacters

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

$	indicates variable substitution
!	indicates history substitution
:	precedes substitution modifiers
↑	used in special forms of history substitution
`	indicates command substitution

Other Metacharacters

#	begins scratch file names; indicates Shell comments
-	prefixes option (flag) arguments to commands
%	prefixes job name specifications

Part II — Programming the Bourne Shell

10. 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

```
tutorial$ tel fred
```

displays those lines in */usr/lib/telnos* that contain the string *fred*. To display those lines containing *fred* followed by those for *bert*, type:

```
tutorial$ tel fred 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:

```
tutorial$ create alpha beta
```

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

11. Control Flow — case

The **case** notation provides a multiple way branch. 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

```
tutorial$ append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command. To append the contents of *file1* onto *file2*, say:

```
tutorial$ append file1 file2
tutorial$
```

If the number of arguments supplied to *append* is other than 1 or 2, a message is displayed 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, you can use it 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 ?.

12. Here Documents

The Shell procedure *tel* in 'Control Flow — for' 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

```
tutorial% 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*. You can prevent substitution 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* displays a ? if there are no occurrences of the string $1.) Quoting the terminating string prevents substitution entirely within a *here* document, 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.

13. Shell Variables

The Shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. You may give variables values by writing, for example:

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

which assigns values to the variables **user**, **box** and **acct**. To set a variable to the null string, you can say:

```
null=
```

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

```
tutorial$ echo $user
fred
```

echos *fred*.

Use variables interactively to provide abbreviations for frequently used strings. For example:

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

moves 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
```

directs the output of *ps* to the file */tmp/psa*, whereas:

```
ps a >$tmpa
```

causes 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 filenames. 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; avoid them in 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/mail/fred
```

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

```
tutorial$ cd /usr/fred/bin
```

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

```
tutorial$ cat wn
```

displays on the screen 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 your login profile.

$PATH A list of directories that contain commands (the *search path*). Each time the Shell executes a command, a list of directories is searched for an executable file. If **$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*.

14. The 'test' Command

Although not part of the Shell, Shell programs use the *test* command. 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
```

15. 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.)

16. 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 'The 'man' Command' section.

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 displayed. 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, 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.

17. 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*.

18. 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 displayed as it is executed. (Try these at the workstation 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 $-.

19. The 'man' Command

The *man* command displays sections of the user's manual. It is called, for example, as

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

In the first the manual section for *sh* is printed. 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.

```
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

20. 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, ...**

You can also use the *set* command to set positional parameters from within a procedure. For example,

```
set - *
```

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

21. Parameter Transmission

When a Shell procedure is called 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 called, 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 calling 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.

22. 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

```
tutorial$ echo $d
```

or

```
tutorial$ echo ${d}
```

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

```
tutorial$ 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

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

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

```
tutorial$ 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 the Shell prints *message* 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.

23. Command Substitution

In a similar way, you can substitute the standard output from a command to parameters. The command *pwd* displays 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 use of string processing commands 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
```

displays the string *main*. The following fragment from a *cc* command illustrates its use:

```
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, such as, *1977 Nov 1, 23:59:59*

24. Evaluation and Quoting

The Shell is a macro processor that provides parameter substitution, command substitution and filename 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 the 'Grammar' section. Before a command is executed, the following substitutions occur.

- Parameter substitution, such as **$user**
- Command substitution, such as **`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.

- Filename 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 filename 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 occur, but filename 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, use the built-in command *eval*. 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.

25. 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 'Command Execution') 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 3 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 goes on to execute the next command. Except for the last case, the Shell displays an error message. 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 such as, 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** terminates the Shell 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.

26. 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 'Command Execution'), then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 5). The cleanup action is to remove the file **junk$$**.

```
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, the Shell uses it 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 both the procedure and the invoked commands to ignore the *hangup*, *interrupt*,and *kill* signals.

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 6) 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.

27. 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 filename 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,

```
tutorial$ 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 the command ignores them. 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.

28. Calling the Shell

The Shell interprets the following flags when it is called. 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, 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, the Shell ignores QUIT.

29. Grammar

Commands are parsed initially according to the following 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**

30. Metacharacters 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
{ }

Table of Contents

UNIX Programming

This chapter is an introduction to programming on the UNIX† system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

This chapter describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of The Sun Reference Manuals (*User's Manual*, *System Interface Manual*, and *System Manager's Manual*)[1]. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

1. Basics

1.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function **main** as an argument count **argc** and an array **argv** of pointers to character strings that contain the arguments. By convention, **argv[0]** is the command name itself, so **argc** is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal — This is essentially the **echo** command.

† UNIX is a trademark of Bell Laboratories.

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by \0, so they can be treated as strings. The program starts by printing **argv[1]** and loops until it has printed them all.

The argument count and the arguments are parameters to **main**. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2. The 'Standard Input' and 'Standard Output'

The simplest input mechanism is to read the 'standard input,' which is generally the user's terminal. The function **getchar** returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if **prog** uses **getchar**, then the command line

tutorial% **prog < file**

causes **prog** to read **file** instead of the terminal. **prog** itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the *pipe* mechanism:

tutorial% **otherprog | prog**

provides the standard input for **prog** from the standard output of **otherprog**.

getchar returns the value **EOF** when it encounters the end of file (or an error) on whatever you are reading. The value of **EOF** is normally defined to be **-1**, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, **putchar(c)** puts the character **c** on the 'standard output,' which is also by default the terminal. The output can be captured on a file by using >: if **prog** uses **putchar**,

tutorial% **prog > outfile**

writes the standard output on **outfile** instead of the terminal. **outfile** is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

tutorial% **prog | otherprog**

puts the standard output of **prog** into the standard input of **otherprog**.

The function **printf**, which formats output in various ways, uses the same mechanism as **putchar** does, so calls to **printf** and **putchar** may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function **scanf** provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. **scanf** uses the same mechanism as

getchar, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with **getchar**, **putchar**, **scanf**, and **printf** may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()      /* ccstrip: strip non-graphic characters */
{
        int c;
        while ((c = getchar()) != EOF)
                if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
                        putchar(c);
        exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of **EOF**.

If it is necessary to treat multiple files, you can use **cat** to collect the files for you:

```
tutorial% cat file1 file2 ... | ccstrip > output
```

and thus avoid learning how to access files from a program. By the way, the call to **exit** at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The 'Standard I/O Library' is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
tutorial% wc x.c y.c
```

prints the number of lines, words and characters in x.c and y.c and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function **fopen**. **fopen** takes an external name (like **x.c** or **y.c**), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including **stdio.h** is a structure definition called **FILE**. The only declaration needed for a file pointer is exemplified by

```
FILE    *fp, *fopen();
```

This says that **fp** is a pointer to a **FILE**, and **fopen** returns a pointer to a **FILE**. **FILE** (is a type name, like **int**, not a structure tag.

The actual call to **fopen** in a program is

```
fp = fopen(name, mode);
```

The first argument of **fopen** is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read "**r**"), (write "**w**"), (or append "**a**"). (

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, **fopen** will return the null pointer value **NULL** (which is defined as zero in **stdio.h**).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which **getc** and **putc** are the simplest. **getc** returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in **c** the next character from the file referred to by **fp**; it returns **EOF** when it reaches end of file. **putc** is the inverse of **getc**:

```
putc(c, fp)
```

puts the character **c** on the file **fp** and returns **c**. **getc** and **putc** return **EOF** on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called **stdin**, **stdout**, and **stderr**. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. **stdin**, **stdout** and **stderr** are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type **FILE** * can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way

the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[ ];
{
        int c, i, inword;
        FILE *fp, *fopen();
        long linect, wordct, charct;
        long tlinect = 0, twordct = 0, tcharct = 0;

        i = 1;
        fp = stdin;
        do {
                if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
                        fprintf(stderr, "wc: can't open %s\n", argv[i]);
                        continue;
                }
                linect = wordct = charct = inword = 0;
                while ((c = getc(fp)) != EOF) {
                        charct++;
                        if (c == '\n')
                                linect++;
                        if (c == ' ' || c == '\t' || c == '\n')
                                inword = 0;
                        else if (inword == 0) {
                                inword = 1;
                                wordct++;
                        }
                }
                printf("%7ld %7ld %7ld", linect, wordct, charct);
                printf(argc > 1 ? " %s\n" : "\n", argv[i]);
                fclose(fp);
                tlinect += linect;
                twordct += wordct;
                tcharct += charct;
        } while (++i < argc);
        if (argc > 2)
                printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
        exit(0);
}
```

The function **fprintf** is identical to **printf**, save that the first argument is a file pointer that specifies the file to be written.

The function **fclose** is the inverse of **fopen**; it breaks the connection between the file pointer and the external name that was established by **fopen**, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call **fclose** on an output file — it flushes the buffer in which **putc** is collecting output. (**fclose** is

called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

stderr is assigned to a program in the same way that **stdin** and **stdout** are. Output written on **stderr** appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on **stderr** instead of **stdout** so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function **exit** to terminate program execution. The argument of **exit** is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

exit itself calls **fclose** for each open output file, to flush out any buffered output, then calls a routine named **_exit**. The function **_exit** causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with **putc**, etc., is buffered (except to **stderr**); to force it out immediately, use **fflush(fp)**.

fscanf is identical to **scanf**, except that its first argument is a file pointer (as with **fprintf**) that specifies the file from which the input comes; it returns **EOF** at end of file.

The functions **sscanf** and **sprintf** are identical to **fscanf** and **fprintf**, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for **sscanf** and into it for **sprintf**.

fgets(buf, size, fp) copies the next line from **fp**, up to and including a newline, into **buf**; at most **size-1** characters are copied; it returns **NULL** at end of file. **fputs(buf, fp)** writes the string in **buf** onto file **fp**.

The function **ungetc(c, fp)** 'pushes back' the character **c** onto the input stream **fp**; a subsequent call to **getc**, **fscanf**, etc., will encounter **c**. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called 'opening' the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the 'shell') runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
tutorial% prog < infile > outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called **read** and **write**. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than n bytes remained to be read. (When the file is a terminal, **read** normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and **-1** indicates an error of some sort. For writing, the returned value is the

number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ('unbuffered'), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512     /* best size for PDP-11 UNIX */

main()      /* copy input to output */
{
        char  buf[BUFSIZE];
        int   n;

        while ((n = read(0, buf, BUFSIZE)) > 0)
                write(1, buf, n);
        exit(0);
}
```

If the file size is not a multiple of **BUFSIZE**, some **read** will return a smaller number of bytes to be written by **write**; the next call to **read** after that will return zero.

It is instructive to see how **read** and **write** can be used to construct higher level routines like **getchar**, **putchar**, etc. For example, here is a version of **getchar** which does unbuffered input.

```
#define CMASK   0377    /* for making char's > 0 */

getchar()   /* unbuffered single character input */
{
        char c;

        return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared **char**, because **read** accepts a character pointer. The character being returned must be masked with **0377** to ensure that it is positive; otherwise sign extension may make it negative. (The constant **0377** is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of **getchar** does input in big chunks, and hands out the characters one at a time.

```
#define   CMASK   0377 /* for making char's > 0 */
#define   BUFSIZE  512

getchar()   /* buffered version */
{
      static char buf[BUFSIZE];
      static char *bufp = buf;
      static int   n = 0;

      if (n == 0) {    /* buffer is empty */
            n = read(0, buf, BUFSIZE);
            bufp = buf;
      }
      return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, **open** and **creat** [sic].

open is rather like the **fopen** discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an **int**.

```
int fd;

fd = open(name, rwmode);
```

As with **fopen**, the **name** argument is a character string corresponding to the external file name. The access mode argument is different, however: **rwmode** is 0 for read, 1 for write, and 2 for read and write access. **open** returns **-1** if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to **open** a file that does not exist. The entry point **creat** is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called **name**, and **-1** if not. If the file already exists, **creat** will truncate it to zero length; it is not an error to **creat** a file that already exists.

If the file is brand new, **creat** creates it with the *protection mode* specified by the **pmode** argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv) /* cp: copy f1 to f2 */
int argc;
char *argv[ ];
{
        int    f1, f2, n;
        char  buf[BUFSIZE];

        if (argc != 3)
                error("Usage: cp from to", NULL);
        if ((f1 = open(argv[1], 0)) == -1)
                error("cp: can't open %s", argv[1]);
        if ((f2 = creat(argv[2], PMODE)) == -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) != n)
                        error("cp: write error", NULL);
        exit(0);
}

error(s1, s2)       /* print error message and die */
char *s1, *s2;
{
        printf(s1, s2);
        printf("\n");
        exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine **close** breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via **exit** or return from the main program closes all open files.

The function **unlink(filename)** removes the file **filename** from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each **read** or **write** takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call **lseek** provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is **fd** to move to position **offset**, which is taken relative to the location specified by **origin**. Subsequent reading or writing will begin at

that position. **offset** is a **long**; **fd** and **origin** are **int**'s. **origin** can be 0, 1, or 2 to specify that **offset** is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ('rewind'),

```
lseek(fd, 0L, 0);
```

Notice the **0L** argument; it could also be written as **(long) 0**.

With **lseek**, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
        lseek(fd, pos, 0); /* get to pos */
        return(read(fd, buf, n));
}
```

In pre-version 7 UNIX the basic entry point to the I/O system was called **seek**. **seek** was identical to **lseek**, except that its **offset** argument was an **int** rather than a **long**. Accordingly, since PDP -11 integers have only 16 bits, the **offset** specified for **seek** was limited to 65,535; for this reason, **origin** values of 3, 4, 5 caused **seek** to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret **origin** as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file required two seeks, first one which selected the block, then one which has **origin** equal to 1 and moved to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell **errno**. The meanings of the various error numbers are listed in *intro*(2) in the Sun *System Interface Manual* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine **perror** will print a message associated with the value of **errno**; more generally, **sys_errno** is an array of character strings which can be indexed by **errno** and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The 'System' Function

The easiest way to execute a program from another is to use the standard library routine **system**. **system** takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main( ) {
    system("date"); /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of **sprintf** may be useful.

Remember than **getc** and **putc** normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use **fflush**; for input, see **setbuf** in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's **system** routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine **execl** . To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to **execl** is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a **NULL** argument.

The **execl** call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an **execl** call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where **date** is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of **execl** called **execv** is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where **argp** is an array of pointers to the arguments; the last pointer in the array must be **NULL** so **execv** can tell where the list ends. As with **execl**, **filename** is the file in which the

program is found, and **argp[0]** is the name of the program. (This arrangement is identical to the **argv** array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use **execl** to invoke the shell **sh**, which then does all the work. Construct a string **commandline** that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, **/bin/sh**. Its argument **-c** says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in **commandline**.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with **execl** or **execv**. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called **fork**:

```
proc_id = fork( );
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of **proc_id**, the 'process id.' In one of these processes (the 'child'), **proc_id** is zero. In the other (the 'parent'), **proc_id** is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork( ) == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);      /* in child */
```

And in fact, except for handling errors, this is sufficient. The **fork** makes two copies of the program. In the child, the value returned by **fork** is zero, so it calls **execl** which does the command and then dies. In the parent, **fork** returns non-zero so it skips the **execl**. (If there is any error, **fork** returns **-1**).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function **wait**:

```
int status;

if (fork( ) == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the **execl** or **fork**, or the possibility that there might be more than one child running simultaneously. (The **wait** returns the process id of the terminated child, if you want to check it against the value returned by **fork** .) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in **status**). Still, these three lines are the heart of the standard library's **system** routine, which we'll show in a moment.

The **status** returned by **wait** encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to **exit** which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither **fork** nor the **exec** calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the **execl**. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
tutorial% ls | pr
```

which connects the standard output of **ls** to the standard input of **pr**. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call **pipe** creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int     fd[2];

stat = pipe(fd);
if (stat == -1)
        /* there was an error ... */
```

fd is an array of two file descriptors, where **fd[0]** is the read side of the pipe and **fd[1]** is for writing. These may be used in **read**, **write** and **close** calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent **read** will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called **popen(cmd, mode)**, which creates a process **cmd** (just as **system** does), and returns a file descriptor that will either read or write that process, according to **mode**. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the **pr** command; subsequent **write** calls using the file descriptor **fout** will send their data to that process through the pipe.

popen first creates the the pipe with a **pipe** system call; it then **forks** to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via **execl**) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will

never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define   READ    0
#define   WRITE   1
#define   tst(a, b)   (mode == READ ? (b) : (a))
static int   popen_pid;

popen(cmd, mode)
char  *cmd;
int   mode;
{
      int p[2];

      if (pipe(p) < 0)
            return(NULL);
      if ((popen_pid = fork( )) == 0) {
            close(tst(p[WRITE], p[READ]));
            close(tst(0, 1));
            dup(tst(p[READ], p[WRITE]));
            close(tst(p[READ], p[WRITE]));
            execl("/bin/sh", "sh", "-c", cmd, 0);
            _exit(1);    /* disaster has occurred if we get here */
      }
      if (popen_pid == -1)
            return(NULL);
      close(tst(p[READ], p[WRITE]));
      return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1)); dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The close closes file descriptor 0, that is, the standard input. dup is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the dup is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. * Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function pclose to close the pipe created by popen. The main reason for using a separate function rather than close is that it is desirable to wait for the termination of the child process. First, the return value from pclose indicates whether the process succeeded. Equally important when a process creates several children is

* Yes, this is a bit tricky, but it's a standard idiom.

that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the wait lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)  /* close pipe fd */
int fd;
{
        register r, (*hstat)( ), (*istat)( ), (*qstat)( );
        int     status;
        extern int popen_pid;

        close(fd);
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);
        while ((r = wait(&status)) != popen_pid && r != -1);
        if (r == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);
        return(status);
}
```

The calls to **signal** make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable **popen_pid**; it really should be an array indexed by file descriptor. A **popen** function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called **signal**. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file **signal.h** gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, **signal** returns the previous value of the signal. The second argument to **signal** may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main( )
{
    int onintr( );

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */

    exit(0);
}

onintr( )
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to **signal**? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the **onintr** routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that **signal** returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main( )
{
        int (*istat)( ), onintr( );

        istat = signal(SIGINT, SIG_IGN);        /* save original status */
        setjmp(sjbuf);    /* save current stack position */
        if (istat != SIG_IGN)
                signal(SIGINT, onintr);

        /* main processing loop */
}

onintr( )
{
        printf("\nInterrupt\n");
        longjmp(sjbuf);  /* return to saved state */
}
```

The include file **setjmp.h** declares the type **jmp_buf** an object in which the state can be saved. **sjbuf** is such an object; it is an array of some sort. The **setjmp** routine then saves the state of things. When an interrupt occurs, a call is forced to the **onintr** routine, which can print a message, set flags, or whatever. **longjmp** takes as argument an object stored into by **setjmp**, and restores control to the location after the call to **setjmp**, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling **exit** or **longjmp**, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that 'execution resumes at the exact point it was interrupted,' the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for 'errors' which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, **wait**, and **pause**). A program whose **onintr** program just sets **intflag**, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar( ) == EOF)
        if (intflag)
                /* EOF caused by interrupt */ else
                /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like '!' in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork( ) == 0)
        execl(...);
signal(SIGINT, SIG_IGN);  /* ignore interrupts */
wait(&status);   /* until the child is done */
signal(SIGINT, onintr);       /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function **system**:

```
#include <signal.h>

system(s)  /* run command string s */
char *s;
{
        int status, pid, w;
        register int (*istat)( ), (*qstat)( );

        if ((pid = fork( )) == 0) {
                execl("/bin/sh", "sh", "-c", s, 0);
                _exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

As an aside on declarations, the function **signal** obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values **SIG_IGN** and **SIG_DFL** have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they

are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define SIG_DFL (int (*)( ))0
#define SIG_IGN (int (*)( ))1
```

7. References

[1] Sun Microsystems Reference Manuals: *User's Manual*, *System Interface Manual*, and *System Manager's Manual*.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[3] B. W. Kernighan, *UNIX for Beginners — Second Edition*, Bell Laboratories, 1978. Reprinted in the Sun *Tutorial for Beginners* Manual.

Appendix A. The Standard I/O Library

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX

A.1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF is actually -1, and is the value returned by the read routines on end-of-file or error.

NULL is a notation for the null pointer, returned by pointer-valued functions to indicate an error

FILE expands to **struct _iob** and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See **setbuf**, below.

getc, getchar, putc, putchar, feof, ferror, fileno
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names **stdin**, **stdout**, and **stderr** are in effect constants and may not be assigned to.

A.2. Calls

FILE *fopen(filename, type) char *filename, *type;

opens the file and, if needed, allocates a buffer for it. **filename** is a character string specifying the name. **type** is a character string (not a single character). It may be "**r**", "**w**", or "**a**" to indicate intent to read, write, or append. The value returned is a file pointer. If it is

9.3. 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.

9.4. 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.

occurred.

fputs(s, ioptr) char *s; FILE *ioptr;

writes the null-terminated string (character array) **s** on the stream **ioptr**. No newline is appended. No value is returned.

ungetc(c, ioptr) FILE *ioptr;

The argument character **c** is pushed back on the input stream named by **ioptr**. Only one character may be pushed back.

printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...)char *s, *format;

printf writes on the standard output. **fprintf** writes on the named output stream. **sprintf** puts characters in the character array (string) named by **s**. The specifications are as described in **printf (3)** in the Sun *System Interface Manual*.

scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;

scanf reads from the standard input. **fscanf** reads from the named input stream. **sscanf** reads from the character string supplied as **s**. **scanf** reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string **format**, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, **EOF** is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;

reads **nitems** of data beginning at **ptr** from file **ioptr**. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the **fopen** call.

fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;

Like **fread**, but in the other direction.

rewind(ioptr) FILE *ioptr;

rewinds the stream named by **ioptr**. It is not very useful except on input, since a rewound output file is still open only for output.

system(string) char *string;

The **string** is executed by the shell as if typed at the terminal.

getw(ioptr) FILE *ioptr;

returns the next word from the input stream named by **ioptr**. **EOF** is returned on end-of-file or error, but since this a perfectly good integer **feof** and **ferror** should be used. A 'word' is 16 bits on the PDP-11.

putw(w, ioptr) FILE *ioptr;

writes the integer **w** on the named output stream.

setbuf(ioptr, buf) FILE *ioptr; char *buf;

setbuf may be used after a stream has been opened but before I/O has started. If **buf** is **NULL**, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char    buf[BUFSIZ];
```

fileno(ioptr) FILE *ioptr;

returns the integer file descriptor associated with the file.

fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;

The location of the next byte in the stream named by **ioptr** is adjusted. **offset** is a long integer. If **ptrname** is 0, the offset is measured from the beginning of the file; if **ptrname** is 1, the offset is measured from the current read or write pointer; if **ptrname** is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non UNIX systems, the offset must be a value returned from **ftell** and the ptrname must be 0).

long ftell(ioptr) FILE *ioptr;

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non UNIX systems the value of this call is useful only for handing to **fseek**, so as to position the file to the same place it was when **ftell** was called.)

getpw(uid, buf) char *buf;

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array **buf**, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

char *malloc(num);

allocates **num** bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. **NULL** is returned if no space is available.

char *calloc(num, size);

allocates space for **num** items each of size **size**. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. **NULL** is returned if no space is available .

cfree(ptr) char *ptr;

Space is returned to the pool used by **calloc**. Disorder can be expected if the pointer was not obtained from **calloc**.

The following are macros whose definitions may be obtained by including **<ctype.h>**.

isalpha(c) returns non-zero if the argument is alphabetic.

isupper(c) returns non-zero if the argument is upper-case alphabetic.

islower(c) returns non-zero if the argument is lower-case alphabetic.

isdigit(c) returns non-zero if the argument is a digit.

isspace(c) returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

ispunct(c) returns non-zero if the argument is any punctuation character, that is, not a space, letter, digit or control character.

isalnum(c) returns non-zero if the argument is a letter or a digit.

isprint(c) returns non-zero if the argument is printable — a letter, digit, or punctuation character.

iscntrl(c) returns non-zero if the argument is a control character.

isascii(c) returns non-zero if the argument is an ascii character, that is, less than octal 0200.

toupper(c) returns the upper-case character corresponding to the lower-case letter c.

tolower(c) returns the lower-case character corresponding to the upper-case letter c.

Table of Contents

LINT — A C PROGRAM CHECKER

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

Lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint* , gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

1. Using Lint

Suppose there are two C[1] source files, ***file1.c*** and ***file2.c***, which are ordinarily compiled and loaded together. Then the command:

```
tutorial% lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command:

```
tutorial% lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

2. A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous 'halting problem,' known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form '*zzz* might be a bug' are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

3. Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These 'errors of commission' rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never referenced; thus the statement:

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the **–x** flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **–v** option is available to suppress the printing of complaints about unused arguments. When **–v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The **–u** flag may be used to suppress the spurious messages which might otherwise appear.

4. Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a 'use,' since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (for example, might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

5. Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,[2] and especially *lex*,[3] may have literally hundreds of unreachable **break** statements. The –O flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the **–b** option.

6. Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function 'values' which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both:

```
return( expr );
```

and:

```
return ;
```

statements is cause for alarm; *lint* will give the message

function *name* contains return(e) and return

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a )
                return ( 3 );
        g ();
}
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint* . If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the 'noise' messages produced by *lint* .

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (for example, not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in 'working' programs; the desired function value just happened to have been computed in the function return register!

7. Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

8. Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

9. Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from -128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment:

```
char c;
...
if( (c = getchar()) < 0 ) ...
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say 'nonportable character comparison'.

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned** .

10. Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int** , which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs** . When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

11. Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint;* the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The –h flag is used to enable these checks. For example, in the statement:

```
*p++ ;
```

the * does nothing; this provokes the message 'null effect' from *lint* . The program fragment:

```
unsigned x ; if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test:

```
if( x > 0 ) ...
```

is equivalent to:

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say 'degenerate unsigned comparison' in these cases. If one says:

```
if( 1 != 0 ) ...
```

lint will report 'constant in conditional context', since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the –h flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

12. Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (for example,, =+ , =–, ...) could cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either:

```
a =- 1 ;
```

or:

```
a =-1;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed:

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example:

```
int x (-1);
```

looks somewhat like the beginning of a function declaration:

```
int x (y){ ...
```

and the compiler must read a fair ways past *x* in order to sure what the declaration really is. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1;
```

This is free of any possible syntactic ambiguity.

13. Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message 'possible pointer alignment problem' results from this situation whenever either the **-p** or **-h** flags are in effect.

14. Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++];
```

will draw the complaint:

```
warning: i evaluation order undefined
```

15. Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler[4, 5] which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint* .

16. Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as:

```
int a;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the **-p** flag, it will detect such multiple definitions.

† UNIX is a trademark of Bell Laboratories.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* –p causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ASCII, while they are eight bit EBCDIC on the IBM, and nine bit ASCII on GCOS. Moreover, character strings go from high to low bit positions ('left to right') on GCOS and IBM, and low to high ('right to left') on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing:

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing:

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

17. Shutting Lint Up

There are occasions when the programmer is smarter than *lint* . There may be valid reasons for 'illegal' type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint* , typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint* , this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The –v flag can be turned on for one function by the directive:

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive:

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive:

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

18. Library Declaration Files

Lint accepts certain library directives, such as:

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

19. Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint* , the desirable properties of universality and portability.

20. References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).
2. S. C. Johnson, 'Yacc: Yet Another Compiler-Compiler,' Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
3. M. E. Lesk, 'Lex — A Lexical Analyzer Generator,' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson and D. M. Ritchie, 'UNIX Time-Sharing System: Portability of C Programs and the UNIX System,' *Bell Sys. Tech. J.* **57**(6) pp. 2021-2048 (1978).
5. S. C. Johnson, 'A Portable Compiler: Theory and Practice,' *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).

Appendix A. Current Lint Options

The command currently has the form

tutorial% **lint [–options] files... library-descriptors...**

The options are

h Perform heuristic checks
p Perform portability checks
v Don't report unused arguments
u Don't report unused or undefined externals
b Report unreachable **break** statements.
x Report unused external declarations
a Report assignments of **long** to **int** or shorter.
c Complain about questionable casts
n No library checking is done
s Same as **h** (for historical reasons)

Table of Contents

MAKE — A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS

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, *Make* 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.

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 (for example, 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:

```
tutorial% 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.

1. 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:

```
tutorial% 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 (*that is*, 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:

tutorial% **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:

tutorial% **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:

tutorial% **make**

loads the three object files with the *lS* library. The command:

tutorial% **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 Bell Laboratories.

2. 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 (for example, *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.

3. Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

tutorial% **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).

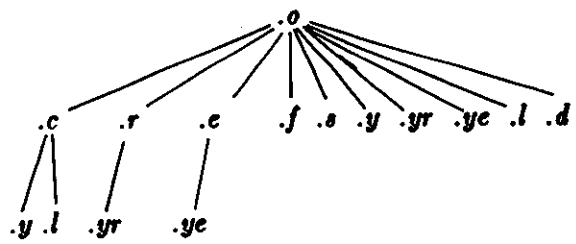
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is 'made'.

4. 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:

Suffix	Type of File
.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:

tutorial% **make CC=newcc**

uses the 'newcc' command 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:

tutorial% **make "CFLAGS=-O"**

uses the optimizing C compiler. causes the optimizing C compiler to be used.

5. 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
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:

```
tutorial% 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:

```
tutorial% make print "P = opr -sp"
```

or:

```
tutorial% make print "P= cat >zap"
```

6. 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:

```
tutorial% 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 (for example, 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:

```
tutorial% 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.

7. 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*.

8. 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.

9. Appendix. Suffixes and Transformation Rules

Make 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 $@
```

Table of Contents

SOURCE CODE CONTROL SYSTEM

The Source Code Control System (SCCS) is a system for controlling changes to text files (typically, the source code and documentation of software systems).

You can think of SCCS as a custodian of files: SCCS provides facilities for storing, updating, and retrieving any version of a file of text; for controlling updating privileges to that file; for identifying the version of a retrieved file; and for recording who made each change, when and where it was made, and why. This is important in environments where programs and documentation undergo frequent changes (due to maintenance and/or enhancement work), because regenerating an unrevised version of a program or document is often desirable. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution to stockpiling multiple versions of the same text, because it stores only the original file and subsequent sets of *changes* on disk.

There are two major divisions of SCCS and these two divisions are reflected in the layout of this document:

- The *sccs* command itself is a high-level 'user friendly' front end that provides an interface to a collection of tools for manipulating SCCS files. In general, users can get by using the facilities provided by the *sccs* command, and so *sccs* is described in Part I of this document. The individual SCCS tools are not too easy to use, but they do provide extremely close control over the SCCS database files.
- The SCCS commands are a collection of programs for manipulating the SCCS database files. Although the *sccs* front end command normally abstracts the most common operations you might want to do, there may be times when it is necessary to use the raw facilities of the SCCS commands themselves — these commands are described in Part II of this document and gives a deeper description of how to use SCCS. Of particular interest are the numbering of branches, the *l-file*, which gives a description of what deltas were used on a *get*, and certain other SCCS commands.

The SCCS manual pages are a good last resort. These should be read by software managers and by people who want to know everything about everything.

Both the *SCCS User's Guide* and the SCCS manual pages were written in the days before the *sccs* command existed, so most of the examples are slightly different from those in this document.

Part I — The SCCS High-Level User Interface

This first part of this document is a quick introduction to using SCCS via the *sccs* command. The presentation is geared towards people who want to know how to get the job done done rather than how the SCCS works; for this reason some of the examples are not well explained. For details of what the magic options do, see the section entitled: *Further Information.*

Throughout this introduction, we assume that you are using the C-Shell on a machine called 'tutorial', and so the hostname is shown followed by the % sign prompt in the examples. What you type is shown in **bold faced text like this**, and the system's responses are shown in ordinary typeface, like this:

```
tutorial% sccs get prog.c
1.1
87 lines
```

SCCS is a source management system. Such a system maintains a record of versions of a system; a record is kept with each set of changes of what the changes are, why they were made, who made them, and when they were made. Old versions can be recovered, and different versions can be maintained simultaneously. In projects with more than one person, ensures that two people are not editing the same file at the same time.

All versions of your program, plus the log and other information, is kept in a file called the *s-file*. There are three major operations that can be performed on the *s-file*:

1. Get a file. This operation retrieves a version of the file from the *s-file*. By default, the latest version is retrieved. This file is intended for compilation, printing, or whatever; it is specifically NOT intended to be edited or changed in any way; any changes made to a file retrieved in this way will probably be lost.
2. Get a file for editing. This operation also retrieves a version of the file from the *s-file*, but this file is intended to be edited and then incorporated back into the *s-file*. Only one person may be editing a file at one time.
3. Merge a file back into the *s-file*. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

1. Learning the Lingo

There are a number of terms that are worth learning before we go any farther.

1.1. S-file

The *s-file* is a single file that holds all the different versions of your file. The *s-file* contains only the differences between versions, rather than the entire text of the new version. This saves disk space and allows selective changes to be removed later. Also included in the *s-file* is some header information for each version, including the comments given by the person who created the version explaining why the changes were made.

1.2. Deltas

Each set of changes to the *s-file* — which is approximately, but not exactly, equivalent to a version of the file — is called a *delta*. Although technically a delta only includes the *changes* made, in practice it is usual for each delta to be made with respect to all the deltas that have occurred before[1]. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes — equivalent to removing your changes later.

1.3. SID's (version numbers)

A SID — SCCS Id — is a number that represents a delta. This is normally a two-part number consisting of a 'release' number and a 'level' number. Normally the release number stays the same, however, it is possible to move into a new release if some major change is being made.

Since all past deltas are normally applied, the SID of the final delta applied can be used to represent a version number of the file as a whole.

1.4. Id keywords

When you get a version of a file with intent to compile and install it that is, something other than edit it), some special keywords that are part of the text of the file are expanded inline by SCCS. These *Id Keywords* can be used to include the current version number or other information into the file. All id keywords are of the form %*x*%, where *x* is an upper case letter. For example, %I% is the SID of the latest delta applied, %W% includes the module name, SID, and a mark that makes it findable by a program, and %G% is the date of the latest delta applied. There are many others, most of which are of dubious usefulness.

When you get a file for editing, the id keywords are not expanded; this is so that after you put them back in to the *s-file*, they will be expanded automatically on each new version. But notice: if you were to get them expanded accidently, your file would appear to be the same version forever more, which would of course defeat the purpose. Also, if you should install a version of the program without expanding the id keywords, it will be impossible to tell what version it is (since all it will have is '%W%' or whatever).

[1] This matches normal usage, where the previous changes are not saved at all, so all changes are automatically based on all other changes that have happened through history.

2. Creating SCCS Database Files with 'sccs create'

To put a bunch of source files into SCCS format, you do the following things:

- Make the SCCS subdirectory if it isn't there already:

```
tutorial% mkdir SCCS          Note that SCCS is upper-case
tutorial%
```

- Then you use the *sccs create* command to actually create the SCCS database files for all the source files you have. Suppose that you want to have all your *.c* and *.h* files under SCCS control:

```
tutorial% sccs create *.[ch]
     lots of messages from SCCS here
tutorial%
```

For each *file* you have, the *sccs create* command does the following things for you:

Creates a file called *s.file* in the SCCS subdirectory,

Renames each *file* by placing a comma in front of the name, so that you end up with files of the form *,file*.

Gets a read-only copy of each *file* by using the *sccs get* command, as described later on.

When you are convinced that SCCS has correctly created the *s-files*, you should remove the files whose names start with commas.

If you want to have id keywords in the files, it is best to put them in before you create the *s-files*. If you do not, *create* will print 'No Id Keywords (cm7)', which is a warning message only.

3. Retrieving Files for Compilation with 'sccs get'

To get a copy of the latest version of a file, run

```
tutorial% sccs get prog.c
```

SCCS will respond:

```
1.1
87 lines
```

meaning that version 1.1 was retrieved[2] and that it has 87 lines. The file *prog.c* is created in the current directory — it is created read-only to remind you that you are not supposed to change it.

This copy of the file should not be changed, since SCCS is unable to merge the changes back into the *s-file*. If you do make changes, they will be lost the next time someone does a *get*.

4. Changing Files (Creating Deltas)

[2] Actually, the SID of the final delta applied was 1.1.

4.1. Retrieving a File for Editing with 'sccs edit'

To edit a source file, you must first get it, requesting permission to edit it[3]. The response will be the same as with *get* except that it also says that a new delta is being created:

```
tutorial% sccs edit prog.c
New delta 1.2
```

You then edit it, using a standard text editor:

```
tutorial% vi prog.c
```

4.2. Merging Changes Back Into the s-file with 'sccs delta'

When the desired changes are made, you can put your changes into the SCCS file using the *delta* command:

```
tutorial% sccs delta prog.c
```

Delta prompts you for 'comments?' before merging the changes in. At this prompt you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash. *Delta* will then type:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged[4]. The *prog.c* file will be removed; it can be retrieved using *get*.

4.3. When to Make Deltas

It is probably unwise to make a delta before every recompilation or test; otherwise, you tend to get a lot of deltas with comments like 'fixed compilation problem in previous delta' or 'fixed botch in 1.3'. However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, delta everything being edited, re-get them, and recompile everything.

4.4. Finding Out What's Going On with 'sccs info'

To find out what files are being edited, type:

---

[3] The *edit* command is equivalent to using the –e flag to *get*, as:
tutorial% sccs get –e prog.c
Keep this in mind when reading other documentation.

[4] Changes to a line are counted as a line deleted and a line inserted.

tutorial% **sccs info**

to display a list of all the files being edited and other information — such as the name of the user who did the edit. Also, the command:

tutorial% **sccs check**

is nearly equivalent to the *info* command, except that it is silent if nothing is being edited, and returns non-zero exit status if anything is being edited. It can thus be used in an 'install' entry in a makefile to abort the install if anything has not been properly deltaed.

If you know that everything being edited should be deltaed, you can use:

tutorial% **sccs delta `sccs tell`**

The *tell* command is similar to *info* except that only the names of files being edited are output, one per line.

All of these commands take a –b flag to ignore 'branches' (alternate versions, described later) and the –u flag to only give files being edited by you. The –u flag takes an optional *user* argument, giving only files being edited by that user. For example:

tutorial% **sccs info –ujohn**

gives a listing of files being edited by john.

4.5. ID keywords

Id keywords can be inserted into your file that will be expanded automatically by *get*. For example, a line such as:

static char SccsId[ ] = "%W%\t%G%";

will be replaced with something like:

static char SccsId[ ] = "@(#)prog.c 1.2 08/29/80";

This tells you the name and version of the source file and the time the delta was created. The string '@(#)' is a special string which signals the beginning of an SCCS Id keyword.

4.5.1. Finding Out What Versions Are Being Run with 'sccs what'

To find out what version of a program is being run, use:

tutorial% **sccs what prog.c /usr/bin/prog**

which will print all strings it finds that begin with '@(#)'. This works on all types of files, including binaries and libraries. For example, the above command will output something like:

```
prog.c:
        prog.c      1.2   08/29/80
/usr/bin/prog:
        prog.c      1.1   02/05/79
```

From this I can see that the source that I have in prog.c will not compile into the same version as the binary in /usr/bin/prog.

4.5.2. Where to Put Id Keywords

ID keywords can be inserted anywhere, including in comments, but Id Keywords that are compiled into the object module are especially useful, since it lets you find out what version of the object is being run, as well as the source. However, there is a cost: data space is used up to store the keywords, and on small address space machines this may be prohibitive.

When you put id keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[ ] = "%W%	%G%";
```

in the file *access.h* and:

```
static char OpsysSid[ ] = "%W%%G%";
```

in the file *opsys.h*. Otherwise, you will get compilation errors because 'SccsId' is redefined. The problem with this is that if the header file is included by many modules that are loaded together, the version number of that header file is included in the object module many times; you may find it more to your taste to put id keywords in header files in comments.

4.6. Keeping SID's Consistent Across Files

With some care, it is possible to keep the SID's consistent in multi-file systems. The trick here is to always *edit* all files at once. The changes can then be made to whatever files are necessary and then all files (even those not changed) are redeltaed. This can be done fairly easily by just specifying the name of the directory that the SCCS files are in:

```
tutorial% sccs edit SCCS
```

which will *edit* all files in that directory. To make the delta, use:

```
tutorial% sccs delta SCCS
```

You will be prompted for comments only once.

4.7. Creating New Releases

When you want to create a new release of a program, you can specify the release number you want to create on the *edit* command. For example:

```
tutorial% sccs edit -r2 prog.c
```

will put the next delta in release two (that is, it will be numbered 2.1). Future deltas will automatically be in release two. To change the release number of an entire system, use:

```
tutorial% sccs edit -r2 SCCS
```

5. Restoring Old Versions

5.1. Reverting to Old Versions

Suppose that after delta 1.2 was stable you made and released a delta 1.3. But this introduced a bug, so you made a delta 1.4 to correct it. But 1.4 was still buggy, and you decided you wanted to go back to the old version. You could revert to delta 1.2 by choosing the SID in a get:

```
tutorial% sccs get -r1.2 prog.c
```

This will produce a version of *prog.c* that is delta 1.2 that can be reinstalled so that work can proceed.

In some cases you don't know what the SID of the delta you want is. However, you can revert to the version of the program that was running as of a certain date by using the -c (cutoff) flag. For example,

```
tutorial% sccs get -c800722120000 prog.c
```

will retrieve whatever version was current as of July 22, 1980 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be equivalently stated:

```
tutorial% sccs get -c"80/07/22 12:00:00" prog.c
```

5.2. Selectively Deleting Old Deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this by *excluding* delta 1.3:

```
tutorial% sccs edit -x1.3 prog.c
```

When delta 1.5 is made, it will include the changes made in delta 1.4, but will exclude the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you want to get rid of 1.3 and 1.4 you can use:

```
tutorial% sccs edit -x1.3-1.4 prog.c
```

which will exclude all deltas from 1.3 to 1.4. Alternatively,

```
tutorial% sccs edit -x1.3-1 prog.c
```

will exclude a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using -x (or -i — see below) there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS always prints out a message telling the range of lines affected; these lines should then be examined very carefully to see if the version SCCS got is ok.

Since each delta (in the sense of 'a set of changes') can be excluded at will, it is most useful to put each semantically-distinct change into its own delta.

6. Auditing Changes

6.1. Displaying Delta Comments with 'sccs prt'

When you created a delta, you presumably gave a reason for the delta to the 'comments?' prompt. To print out these comments later, use:

```
tutorial% sccs prt prog.c
```

This will produce a report for each delta of the SID, time and date of creation, user who created the delta, number of lines inserted, deleted, and unchanged, and the comments associated with the delta. For example, the output of the above command might be:

```
D 1.2 80/08/29 12:35:31    bill  2    1    00005/00003/00084
removed "-q" option
D 1.1 79/02/05 00:19:31    eric  1    0    00087/00000/00000
date and time created 80/06/10 00:19:31 by eric
```

6.2. Finding Why Lines Were Inserted

To find out why you inserted lines, you can get a copy of the file with each line preceded by the SID that created it:

```
tutorial% sccs get -m prog.c
```

You can then find out what this delta did by printing the comments using *prt*.

To find out what lines are associated with a particular delta, 1.3 for instance, use:

```
tutorial% sccs get -m -p prog.c | grep '^1.3'
```

The -p flag causes SCCS to output the generated source to the standard output rather than to a file.

6.3. Discovering What Changes You Have Made with 'sccs diffs'

When you are editing a file, you can find out what changes you have made using:

```
tutorial% sccs diffs prog.c
```

Most of the "diff" flags can be used. To pass the -c flag, use -C.

To compare two versions that are in deltas, use:

```
tutorial% sccs sccsdiff -r1.3 -r1.6 prog.c
```

to see the differences between delta 1.3 and delta 1.6.

7. Shorthand Notations

There are several sequences of commands that are used frequently. *Sccs* tries to make it easy to do these.

7.1. Making a Delta and Getting a File with 'sccs delget'

A frequent requirement is to make a delta of some file and then get that file. This is done by using:

```
tutorial% sccs delget prog.c
```

which is entirely equivalent to using:

```
tutorial% sccs delta prog.c
tutorial% sccs get prog.c
```

except that if an error occurs while making a delta of *any* of the files, none of them will be gotten. The *deledit* command is equivalent to *delget* except that the *edit* command is used instead of the *get* command.

7.2. Replacing a Delta with the 'sccs fix'

Frequently, there are small bugs in deltas, for instance, compilation errors, for which there is no reason to maintain an audit trail. To *replace* a delta, use:

```
tutorial% sccs fix -r1.4 prog.c
```

This gets a copy of delta 1.4 of prog.c for you to edit and then deletes delta 1.4 from the SCCS file. When you do a delta of prog.c, it will be delta 1.4 again. The -r flag must be specified, and the delta that is specified must be a leaf delta, that is, no other deltas may have been made subsequent to the creation of that delta.

7.3. Backing Off From an Edit with 'sccs unedit'

If you found you edited a file that you did not want to edit, you can back out by using:

```
tutorial% sccs unedit prog.c
```

7.4. Working From Other Directories with the -d Flag

If you are working on a project where the SCCS code is in a directory somewhere, you may be able to simplify things by using a shell alias. For example, the alias:

```
alias syssccs sccs -d/usr/src
```

will allow you to issue commands such as:

```
syssccs edit cmd/who.c
```

which will look for the file '/usr/src/cmd/SCCS/who.c'. The file 'who.c' is always created in your current directory regardless of the value of the -d flag.

8. Using SCCS on a Project

Working on a project with several people has its own set of special problems. The main problem occurs when two people modify a file at the same time. SCCS prevents this by locking an *s-file* while it is being edited.

As a result, files should not be reserved for editing unless they are actually being edited at the time, since this will prevent other people on the project from making necessary changes. For example, a good scenario for working might be:

```
tutorial% sccs edit a.c g.c t.c
tutorial% vi a.c g.c t.c
# do testing of the (experimental) version
tutorial% sccs delget a.c g.c t.c
tutorial% sccs info
# should respond "Nothing being edited"
tutorial% make install
```

As a general rule, all source files should be deltaed before installing the program for general use. This will ensure that it is possible to restore any version in use at any time.

9. Saving Yourself

9.1. Recovering a Munged Edit File

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[5]. Unfortunately, you can't just remove it and re-*edit* it; SCCS keeps track of the fact that someone is trying to edit it, so it won't let you do it again. Neither can you just get it using *get*, since that would expand the Id keywords. Instead, you can say:

```
tutorial% sccs get -k prog.c
```

This will not expand the Id keywords, so it is safe to do a delta with it.

Alternatively, you can *unedit* and *edit* the file.

9.2. Restoring the s-file

In particularly bad circumstances, the SCCS file itself may get munged. The most common way this happens is that it gets edited. Since SCCS keeps a checksum, you will get errors every time you read the file. To fix this checksum, use:

```
tutorial% sccs admin -z prog.c
```

10. Managing SCCS Files with 'sccs admin'

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the *-f* option. For example:

```
tutorial% sccs admin -fd1 prog.c
```

sets the 'd' flag to the value '1'. This flag can be deleted by using:

[5] Or given up and decided to start over.

```
tutorial% sccs admin -dd prog.c
```

The most useful flags are:

b Allow branches to be made using the -b flag to *edit*.

d*SID*
Default SID to be used on a *get* or *edit*. If this is just a release number it constrains the version to a particular release only.

i Give a fatal error if there are no Id Keywords in a file. This is useful to guarantee that a version of the file does not get merged into the *s-file* that has the Id Keywords inserted as constants instead of internal forms.

y The 'type' of the module. Actually, the value of this flag is unused by SCCS except that it replaces the %Y% keyword.

The -t*file* flag can be used to store descriptive text from *file*. This descriptive text might be the documentation or a design and implementation document. Using the -t flag ensures that if the SCCS file is sent, the documentation will be sent also. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use 'prt -t'.

The *admin* command can be used safely any number of times on files. A file need not be gotten for *admin* to work.

11. Maintaining Different Versions (Branches)

Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a 'branch'. Normally deltas continue in a straight line, each depending on the delta before. Creating a branch 'forks off' a version of the program.

The ability to create branches must be enabled in advance using:

```
tutorial% sccs admin -fb prog.c
```

The -fb flag can be specified when the SCCS file is first created.

11.1. Creating a Branch

To create a branch, use:

```
tutorial% sccs edit -b prog.c
```

This will create a branch with (for example) SID 1.5.1.1. The deltas for this version will be numbered 1.5.1.*n*.

11.2. Getting From a Branch

Deltas in a branch are normally not included when you do a get. To get these versions, you will have to say:

```
tutorial% sccs get -r1.5.1 prog.c
```

11.3. Merging a Branch Back into the Main Trunk

At some point you will have finished the experiment, and if it was successful you will want to incorporate it into the release version. But in the meantime someone may have created a delta 1.6 that you don't want to lose. The commands:

```
tutorial% sccs edit -i1.5.1.1-1.5.1 prog.c
tutorial% sccs delta prog.c
```

will merge all of your changes into the release system. If some of the changes conflict, get will print an error. The generated result should be carefully examined before the delta is made.

11.4. A More Detailed Example

The following technique might be used to maintain a different version of a program. First, create a directory to contain the new version:

```
tutorial% mkdir ../newxyz
tutorial% cd ../newxyz
```

Edit a copy of the program on a branch:

```
tutorial% sccs -d../xyz edit -b prog.c
```

When using the old version, be sure to use the -b flag to info, check, tell, and clean to avoid confusion. For example, use:

```
tutorial% sccs info -b
```

when in the 'xyz' directory.

If you want to save a copy of the program (still on the branch) back in the *s-file*, you can use:

```
tutorial% sccs -d../xyz deledit prog.c
```

which will do a delta on the branch and reedit it for you.

When the experiment is complete, merge it back into the *s-file* using delta:

```
tutorial% sccs -d../xyz delta prog.c
```

At this point you must decide whether this version should be merged back into the trunk, that is, the default version, which may have undergone changes. If so, it can be merged using the -i flag to *edit* as described above.

11.5. A Warning

Branches should be kept to a minimum. After the first branch from the trunk, SID's are assigned rather haphazardly, and the structure gets complex fast.

12. Using SCCS with Make

SCCS and *make* can be made to work together with a little care. A few sample makefiles for common applications are shown.

There are a few basic entries that every makefile ought to have. These are:

a.out
(or whatever the makefile generates.) This entry regenerates a program. If the makefile regenerates many things, this should be called 'all' and should in turn have dependencies on everything the makefile can generate.

install
Moves the objects to the final resting place, doing any special *chmod*'s or *ranlib*'s as appropriate.

sources
Creates all the source files from SCCS files.

clean
Removes all cruft from the directory.

print
Prints the contents of the directory.

The examples shown below are only partial examples, and may omit some of these entries when they are deemed to be obvious.

The *clean* entry should not remove files that can be regenerated from the SCCS files. It is sufficiently important to have the source files around at all times that the only time they should be removed is when the directory is being mothballed. To do this, the command:

```
tutorial% sccs clean
```

can be used. This will remove all files for which an *s-file* exists, but which is not being edited.

12.1. Maintaining Single Programs

Frequently there are directories with several largely unrelated programs (such as simple commands). These can be put into a single makefile:

```
LDFLAGS= -i -s
prog: prog.o
	$(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h
example: example.o
	$(CC) $(LDFLAGS) -o example example.o
example.o: example.c
.DEFAULT:
	sccs get $<
```

The trick here is that the .DEFAULT rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the .o file on the .c file is important. Another way of doing the same thing is:

```
SRCS=    prog.c prog.h example.c
LDFLAGS= -i -s
prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h
example: example.o
        $(CC) $(LDFLAGS) -o example example.o
sources: $(SRCS)
$(SRCS):
        sccs get $@
```

There are a couple of advantages to this approach: (1) the explicit dependencies of the .o on the .c files are not needed, (2) there is an entry called "sources" so if you want to get all the sources you can just say 'make sources' and (3) the makefile is less likely to do confusing things since it won't try to *get* things that do not exist.

12.2. Maintaining A Library

Libraries that are largely static are best updated using explicit commands, since *make* doesn't know about updating them properly. However, libraries that are in the process of being developed can be handled quite adequately. The problem is that the .o files have to be kept out of the library as well as in the library.

```
# configuration information
OBJS=    a.o b.o c.o d.o
SRCS=    a.c b.c c.c d.s x.h y.h z.h
TARG=    /usr/lib
# programs
GET=     sccs get
REL=
AR=      -ar
RANLIB=ranlib
lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)
        $(RANLIB) lib.a
install: lib.a
        sccs check
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a
sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
print: sources
        pr *.h *.[cs]
clean:
        rm -f *.o
        rm -f core a.out $(LIB)
```

The '$(REL)' in the get can be used to get old versions easily; for example:

```
make b.o REL=-r1.3
```

The *install* entry includes the line 'sccs check' before anything else. This guarantees that all the *s-files* are up to date (that is, nothing is being edited), and will abort the *make* if this condition is not met.

12.3. Maintaining A Large Program

```
OBJS=   a.o b.o c.o d.o
SRCS=   a.c b.c y.c d.s x.h y.h z.h
GET=    sccs get
REL=
a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)
sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) $@
```

The *print* and *clean* entries are identical to the previous case. This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules will be recompiled if header files change.

Since *make* does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

The *touch* command brings the modification date of z.h in line with the modification date of x.h. When you have a makefile such as above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on z.h.

Quick Reference

13. Commands

The following commands should all be preceded with 'sccs'. This list is not exhaustive; for more options see *Further Information.*

get Gets files for compilation (not for editing). Id keywords are expanded.

-r*SID*	Version to get.
-p	Send to standard output rather than to the actual file.
-k	Don't expand id keywords.
-i*list*	List of deltas to include.
-x*list*	List of deltas to exclude.
-m	Precede each line with SID of creating delta.
-c*date*	Don't apply any deltas created after *date.*

edit

Gets files for editing. Id keywords are not expanded. Should be matched with a *delta* command.

-r*SID*	Same as *get.* If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID.*
-b	Create a branch.
-i*list*	Same as *get.*
-x*list*	Same as *get.*

delta Merge a file gotten using *edit* back into the *s-file*. Collect comments about why this delta was made.

unedit Remove a file that has been edited previously without merging the changes into the *s-file*.

prt Produce a report of changes.

-t	Print the descriptive text.
-e	Print (nearly) everything.

info Give a list of all files being edited.

-b	Ignore branches.
-u[*user*]	Ignore files not being edited by *user.*

check Same as *info,* except that nothing is printed if nothing is being edited and exit status is returned.

tell Same as *info,* except that one line is produced per file being edited containing only the file name.

clean Remove all files that can be regenerated from the *s-file*.

what Find and print id keywords.

admin Create or set parameters on *s-files*.

	–i*file*	Create, using *file* as the initial contents.
	–z	Rebuild the checksum in case the file has been trashed.
	–f*flag*	Turn on the *flag*.
	–d*flag*	Turn off (delete) the *flag*.
	–t*file*	Replace the descriptive text in the *s-file* with the contents of *file*. If *file* is omitted, the text is deleted. Useful for storing documentation or 'design & implementation' documents to ensure they get distributed with the *s-file*.
		Useful flags are:
	b	Allow branches to be made using the –b flag to *edit*.
	d*SID*	Default SID to be used on a *get* or *edit*.
	i	Cause 'No Id Keywords' error message to be a fatal error rather than a warning.
	t	The module 'type'; the value of this flag replaces the %Y% keyword.
fix	Remove a delta and reedit it.	
delget	Do a *delta* followed by a *get*.	
deledit	Do a *delta* followed by an *edit*.	

14. Id Keywords

%Z%	Expands to '@(#)' for the *what* command to find.
%M%	The current module name, for example, 'prog.c'.
%I%	The highest SID applied.
%W%	A shorthand for '%Z%%M% <tab> %I%'.
%G%	The date of the delta corresponding to the '%I%' keyword.
%R%	The current release number, that is, the first component of the '%I%' keyword.
%Y%	Replaced by the value of the t flag (set by *admin*).

Part II — The SCCS Low-Level Command Interface

Part I of this document described the *sccs* front-end command for using the facilities of SCCS. In general, you can do most things using the *sccs* command, and so you should in theory never have to look at this part of the document. There may be times however, when it is necessary to use the raw facilities of the SCCS commands themselves, and so this part of the document is a reference guide for SCCS. The following topics are covered in this document:

- How to get started with SCCS.
- The scheme used to identify versions of text kept in an SCCS file.
- Basic information needed for day-to-day use of SCCS commands, including a discussion of the more useful arguments.
- Protection and auditing of SCCS files, including the differences between the use of SCCS by *individual* users on one hand, and *groups* of users on the other.

This document is a user's guide to SCCS. This document contains the following sections:

- *SCCS for Beginners:* How to make an SCCS file, update it, and retrieve a version of it.
- *SCCS File Numbering Conventions:* How versions of SCCS files are numbered and named.
- *SCCS Command Conventions:* Conventions and rules generally applicable to all SCCS commands.
- *SCCS Commands:* Explanation of all SCCS commands, with discussions of the more useful arguments.
- *SCCS Files:* Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a 'project SCCS administrator' is introduced.

15. SCCS For Beginners

We assume here that you know how to log onto a UNIX† system, create files, and use a text editor like *ex* or *vi*. If you need more information on these subjects, see the *User's Manual for the Sun UNIX System.*

In this section, we present some basic concepts of SCCS. Examples are fragments of terminal sessions, with what you type shown in **boldface text like this**, and what the terminal displays shown in ordinary Roman text, like the ordinary text of this paragraph. After familiarizing yourself with basics, use the manual pages for detailed SCCS command descriptions.

[5] † UNIX is a trademark of Bell Laboratories.

Note that all the SCCS commands described here live in the */usr/sccs* directory, so you must either state that directory explicitly when using SCCS commands, or include that pathname in your *.login* file. All the examples shown in this guide assume that you have */usr/sccs* in your path and so you just have to type the required SCCS command name.

15.1. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file; each set of changes usually depends on all previous sets. Each set of changes is called a 'delta' and is assigned a name called the *S*CCS *ID*entification string (SID).

The SID is composed of at most four components; for now let's focus on only the first two: the 'release' and 'level' numbers. Each set of changes to a file is named '*release.level*'; hence, the first delta is called '1.1', the second '1.2', the third '1.3', and so on. The release number can also be changed, allowing, for example, deltas '2.1', '3.19', etc. A change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines the version of the SCCS file obtained by applying the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order, to the null (empty) version of the file.

15.2. Creating an SCCS File with 'admin'

Consider, for example, a file called 'lang' containing a list of programming languages:

```
tutorial% cat lang
c
pl/i
fortran
cobol
algol
tutorial%
```

We wish to give SCCS custody of 'lang' by using *admin* (which *administers* SCCS files) to create an SCCS file and initialize delta 1.1. To do so, we use *admin* as shown, and *admin* responds with a message:

```
tutorial% admin -ilang s.lang
No id keywords (cm7)
tutorial%
```

All SCCS files *must* have names that begin with 's.', hence, 's.lang'. The –i option, together with its value 'lang', indicates that *admin* is to create a new SCCS file and *initialize* it with the contents of the file 'lang'. This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The message is a warning message (which may also be issued by other SCCS commands) that you can ignore for the present. In the following examples, this warning message is not shown, although it may actually be issued by the various commands.

Remove the file 'lang' now — it can be easily reconstructed with the *get* command, described below.

15.3. Retrieving a File with 'get'

Get creates (retrieves) the latest version of an SCCS file and gives you some information about it. For example, here is how to retrieve the file we created above:

```
tutorial% get s.lang
1.1
5 lines
tutorial%
```

Get tells you it has retrieved version 1.1 of the file, which contains 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the 's.' prefix from the name of the SCCS file; hence, the file 'lang' is created.

The above *get* command simply creates the read-only file 'lang' and keeps no information whatsoever regarding its creation. If you wish to subsequently change an SCCS file with the *delta* command (see below), however, you must create a file which can be written as well as read. You do this by using *get* with the –e (edit) option:

```
tutorial% get –e s.lang
1.1
new delta 1.2
5 lines
tutorial%
```

When you use the –e option, *get* creates a file 'lang' for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the *p-file*, that the *delta* command reads later. *Get* prints the same messages as before, and in addition displays the SID of the version to be created using *delta*.

You can now change 'lang' by adding (say) **snobol** and **ratfor** to the list using your favorite editor. Then take a look at the new file:

```
tutorial% cat lang
c
pl/i
fortran
cobol
algol
snobol
ratfor
tutorial%
```

15.4. Recording Changes with 'delta'

To record the changes that were applied to 'lang' within the SCCS file, use the *delta* command. *Delta* asks for comments describing the change, and you respond with a description of why the changes were made:

```
tutorial% delta s.lang
comments? added snobol and ratfor
    More messages from delta — see below
tutorial%
```

Delta then reads the *p-file* and determines what changes were made to the file 'lang'. *Delta* does this by doing its own *get* to retrieve the original version, and then applying *diff*(1) to the original version and the edited version. When the changes to 'lang' have been stored in 's.lang', the dialogue with *delta* looks like:

```
tutorial% delta s.lang
comments? added snobol and ratfor
1.2
2 inserted
0 deleted
5 unchanged
tutorial%
```

The number '1.2' is the name of the delta just created, and the next three lines are a summary of the changes made to 's.lang'.

15.5. More about the 'get' Command

As we have seen:

```
tutorial% get s.lang
```

retrieves the latest version (now 1.2) of the file 's.lang' by starting with the original version of the file and successively applying deltas (the changes) in order, until all deltas have been applied. For our example, the following commands are all equivalent:

```
tutorial% get s.lang
tutorial% get -r1 s.lang
tutorial% get -r1.2 s.lang
```

The numbers following the **-r** option are SIDs. Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the *highest* level number that exists within the specified release. Thus, the second *get* retrieves the latest version in release 1, namely 1.2. The third *get* specifically retrieves a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the *release* number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with a **get -r** command to indicate that a new release will be made:

```
tutorial% get -e -r2 s.lang
1.2
new delta 2.1
7 lines
tutorial%
```

Release 2 does not exist, as indicated by the 'new delta' message, so *get* retrieves the latest version *before* release 2. *Get* also changes the release number of the delta we wish to create to 2, and thus names the new version 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*.

Now suppose you edit the file and remove **cobol** from the list of languages, so that the new file looks like this:

```
tutorial% cat lang
c
pl/i
fortran
algol
snobol
ratfor
tutorial%
```

and then use *delta*, you will see from *delta's* output, that version 2.1 is indeed created:

```
tutorial% delta s.lang
comments? deleted cobol from list of languages
2.1
0 inserted
1 deleted
6 unchanged
tutorial%
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner.

15.6. Getting Explanations of Errors with 'help'

Help displays explanations of SCCS commands and diagnostic messages. As an example, let's type a command line incorrectly and generate an error message:

```
tutorial% get abc
ERROR [abc]: not an SCCS file (co1)
tutorial%
```

The string 'co1' is a code for the diagnostic message. Use it as an argument to *help* to get a fuller explanation of the error:

```
tutorial% help co1
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
tutorial%
```

Thus, *help* is useful whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

16. SCCS File Numbering Conventions

You can think of the deltas applied to an SCCS file as the nodes of a tree; the root is the initial version of the file. The root delta (node) is normally named '1.1' and successor deltas (nodes) are named '1.2', '1.3', etc. We have already discussed these two components of the names of the deltas, the 'release' and 'level' numbers; and you have seen that normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, you have seen how to change the release number when

making a delta, to indicate that a major change to the file is being made. The new release number applies to all successor deltas, unless it is specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 1.

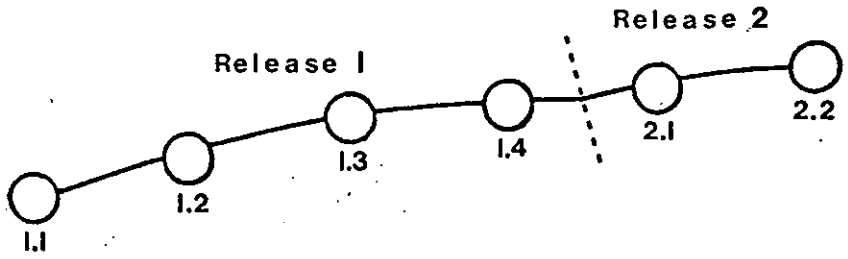


Figure 1: Evolution of an SCCS File

We can call this structure the 'trunk' of the SCCS tree. It represents the normal sequential development of an SCCS file, in which changes that are part of any given delta are dependent upon all the preceding deltas.

However, there are situations when a *branch* on the tree is needed: when changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3 which cannot wait until release 2 to be repaired. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (that is, deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a 'branch' of the tree, and its name consists of four components: the release and level numbers, as with trunk deltas, plus the 'branch' and 'sequence' numbers. Its SID thus appears as: *release.level.branch.sequence*. The *branch* number is assigned to each branch that is a descendant of a particular trunk delta; the first such branch is 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 2.

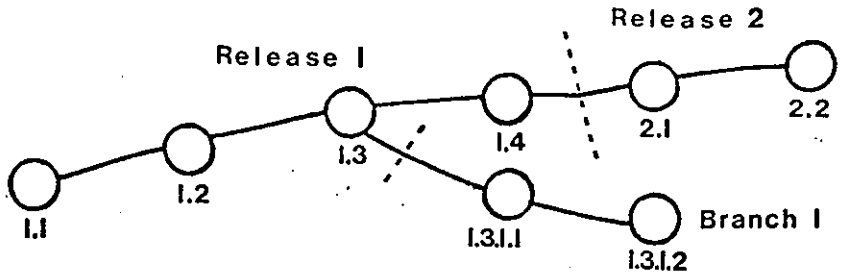


Figure 2: Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the *entire* path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from *it*, all deltas on the new branch will be named 1.3.2.*n* (see Figure 3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the *chronologically* second delta on the *chronologically* second branch whose *trunk* ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

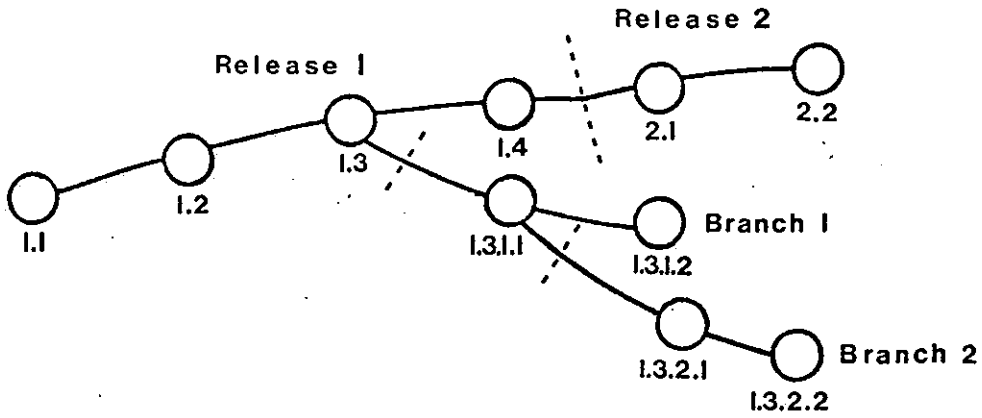


Figure 3: Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

17. SCCS Command Conventions

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below.

17.1. Command Line Syntax

SCCS commands accept *options* and *file arguments*.

Options begin with a minus sign (-), followed by a lower-case alphabetic character, and, in some cases, followed by a value. Options modify actions of commands on which they are specified.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming *all* the SCCS files within the directory. Non- SCCS files and unreadable[6] files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name '-' (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the *name* of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the *find*(1) or *ls*(1) commands. Again, names of non- SCCS files and of unreadable files are silently ignored.

Options specified for a given command apply to *all* file arguments of that command. Options are processed before any file arguments; therefore the placement of options is arbitrary, that is, options may be interspersed with file arguments. File arguments, however, are processed left to right.

[6] Because of permission modes — see *chmod*(1).

Somewhat different argument conventions apply to the *help*, *what*, *sccsdiff*, and *val* commands.

17.2. Flags

Certain actions of various SCCS commands are modified by *flags* embedded in the text of SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see *admin*(1).

17.3. Real/Effective User

The distinction between the *real user* (see *passwd*(1)) and the *effective user* of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same, that is, the user who is logged into a UNIX system.

17.4. Back-up Files Created During Processing

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file*, to ensure that the SCCS file will not be damaged if processing terminates abnormally. The name of the *x-file* is formed by replacing the 's.' of the SCCS file name with 'x.'. When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, is given the same mode (see *chmod*(1)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file*, called the *z-file*, whose name is formed by replacing the 's.' of the SCCS file name with 'z.'. The *z-file* contains the *process number* of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. The *z-file* exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files;* they may be useful in the event of system crashes or similar situations.

17.5. Diagnostics

SCCS commands direct their diagnostic responses to the standard error file. SCCS diagnostics generally look like this:

ERROR [name-of-file-being-processed]: message text (code)

The *code* in parentheses may be used as an argument to *help* to obtain a further explanation of the diagnostic message.

If the SCCS command detects a fatal error during the processing of a file it terminates processing of *that* file and proceeds with the next file in the series, if more than one file has been named.

18. SCCS Commands

This section describes the major features of all the SCCS commands. For detailed descriptions of the commands and of all their arguments, see the individual SCCS manual pages. The discussion below covers only the more common arguments of the various SCCS commands.

The *get* and *delta* commands are presented first because they are the most frequently used. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files; that is, *delta* creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user-specified format.
help	Explains SCCS commands and diagnostic messages.
rmdel	Removes a delta from an SCCS file; useful for removing deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches UNIX file(s) for all occurrences of a special pattern and prints what follows it. *What* is useful in finding identifying information inserted by *get*.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta.
val	Validates an SCCS file.

18.1. get — Retrieve a File

Get creates a text file containing a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file;* its name is formed by removing the 's.' from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The permissions (mode) assigned to the *g-file* depend on the options used with *get*, as discussed below.

Get is normally used to retrieve the latest version of a file on the trunk of the SCCS file tree:

```
tutorial% get s.abc
1.3
67 lines
No id keywords (cm7)
tutorial%
```

The messages tell you that:

1. Version 1.3 of file 's.abc' was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file — see below for a discussion of ID keywords.

The generated *g-file* (file 'abc') is given mode 444 (read-only), since this particular way of invoking *get* is intended to produce *g-files* only for inspection, compilation, or whatever, but *not* for

editing — that is, *not* for making deltas.

If you use *get* with several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it:

```
tutorial% get s.abc s.def
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
tutorial%
```

18.1.1. ID Keywords

When you generate a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file*, so that this information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification* (ID) *keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords.

The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example, %I% is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is defined as the ID keyword for the current date (in the form 'mm/dd/yy'), and %M% is defined as the name of the *g-file*.

Thus, using *get* on an SCCS file that contains the C declaration:

```
char identification [ ] = "%M% %I% %H%";
```

gives (for example) the following:

```
char identification [ ] = "modulename 2.3 03/17/83";
```

If there are no ID keywords in the text, *get* might display:

```
No id keywords (cm7)
tutorial%
```

This message is normally treated as a warning by *get*. However, if an **i** flag is present in the SCCS file, it is treated as an error — see the section entitled *delta — Make a Delta* for further information).

For a complete list of the approximately twenty ID keywords provided, see *get*(1).

18.1.2. Retrieving Different Versions

You can retrieve versions other than the default version of an SCCS file by using various options. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag,

the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **-r** option of *get*.

The **-r** option specifies an SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example, to retrieve version 1.3 of file 's.abc', type:

```
tutorial% get -r1.3 s.abc
1.3
64 lines
tutorial%
```

A branch delta may be retrieved in the same way:

```
tutorial% get -r1.5.2.3 s.abc
1.5.2.3
234 lines
tutorial%
```

When a two- or four-component SID is specified as a value for the **-r** option (as above) and the particular version does not exist in the SCCS file, an error message results.

If you omit the level number of the SID, *get* retrieves the *trunk* delta with the highest level number within the given release, if the given release exists:

```
tutorial% get -r3 s.abc
3.7
213 lines
tutorial%
```

Get retrieved delta 3.7, the highest level trunk delta in release 3. If the given release does not exist, *get* goes to the next-highest existing release, and retrieves the *trunk* delta with the highest level number. For example, if release 9 does not exist in file 's.abc', and release 7 is actually the highest-numbered release below 9, then *get* would generate:

```
tutorial% get -r9 s.abc
7.6
420 lines
tutorial%
```

indicating that trunk delta 7.6 is the latest version of file 's.abc' below release 9.

Similarly, if you omit the sequence number of a SID, as in:

```
tutorial% get -r4.3.2 s.abc
4.3.2.8
89 lines
tutorial%
```

get retrieves the branch delta with the highest sequence number on the given branch, if it exists. If the given branch does not exist, an error message results.

The **-t** option retrieves the latest ('top') version in a particular *release* (that is, when no **-r** option is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is trunk delta 3.5, doing a **get -t** on release 3 produces:

```
tutorial% get -r3 -t s.abc
3.5
59 lines
tutorial%
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command produces:

```
tutorial% get -r3 -t s.abc
3.2.1.5
46 lines
tutorial%
```

18.1.3. Retrieving to Make Changes

Specifying the **-e** option to the *get* command indicates the intent to make a delta sometime later, and, as such, its use is restricted. If the **-e** option is present, *get* checks the following things:

1. The *user list*, the list of *login* names and/or *group IDs* of users allowed to make deltas, to determine if the login name or group ID of the user executing *get* is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.
2. That the *release* (R) of the version being retrieved satisfies the relation:

   $$\text{floor} \leq R \leq \text{ceiling}$$

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.
3. That the *release* (R) is not *locked* against editing. The *lock* is specified as a flag in the SCCS file.
4. Whether or not *multiple concurrent edits* are allowed for the SCCS file as specified by the **j** flag in the SCCS file. Multiple concurrent edits are described in the section entitled *Concurrent Edits of the Same SID*.

Get terminates processing of the corresponding SCCS file if any of the first three conditions fail.

If the above checks succeed, *get* with the **-e** option creates a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user.

Get terminates with an error if a *writable g-file* already exists — this is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

ID keywords appearing in the *g-file* are *not* substituted by *get* when the **-e** option is specified, because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would permanently change them within the SCCS file. In view of this, *get* does not check for the presence of ID keywords within the *g-file*, so that the message: 'No id keywords (cm7)' is never displayed when *get* is invoked with the **-e** option.

In addition, a *get* with the **-e** option creates (or updates) a *p-file*, for passing information to the *delta* command. Let's look at an example of **get -e**:

```
tutorial% get -e s.abc
1.3
new delta 1.4
67 lines
tutorial%
```

The message indicates that *get* has retrieved version 1.3, which has 67 lines; the version *delta* will create is version 1.4.

If the **-r** and/or **-t** options are used together with the **-e** option, the version retrieved for editing is as specified by the **-r** and/or **-t** options.

The options **-i** and **-x** may be used to specify a list of deltas to be *included* and *excluded*, respectively, by *get*. See *get*(1) for the syntax of such a list. 'Including a delta' forces the changes that constitute the particular delta to be included in the retrieved version — this is useful for applying the same changes to more than one version of the SCCS file. 'Excluding a delta' forces it *not* to be applied. This is useful for undoing the effects of a previous delta in the version of the SCCS file to be created.

Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that displays the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists, and to take whatever corrective measures are deemed necessary.

⇒ *The* **-i** *and* **-x** *options should be used with extreme care.*

The **-k** option to *get* can be used to regenerate a *g-file* that may have been accidentally removed or ruined after executing *get* with the **-e** option, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **-k** option is identical to one produced by *get* executed with the **-e** option. However, no processing related to the *p-file* takes place.

18.1.4. Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be 'in progress' at any given time. In general, several people may simultaneously edit the same SCCS file provided they are editing *different versions* of that file. This is the situation we discuss in the following section. However, there is a provision for multiple concurrent edits, so that more than one person can edit the *same version* — see the section entitled *Concurrent Edits of the Same SID*.

The *p-file* — created via a **get -e** command — is named by replacing the 's.' in the SCCS file name with 'p.'. The *p-file* is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The *p-file* contains the following information for each delta that is still 'in progress':[7]

- The SID of the retrieved version.

[7] Other information may be present, but is not of concern here. See *get*(1) for further discussion.

- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get*.

The first execution of **get –e** *creates* the *p-file* for the corresponding SCCS file. Subsequent executions only *update* the *p-file* by inserting a line containing the above information. Before inserting this line, however, *get* performs two checks. First, it searches the entries in the *p-file* for an SID which matches that of the requested version, to make sure that the requested version has not already been retrieved. Secondly, *get* determines whether or not multiple concurrent edits are allowed. If the requested version has been retrieved and multiple concurrent edits are not allowed, an error message results. Otherwise, the user is informed that other deltas are in progress, and processing continues.

It is important to note that the various executions of *get* should be carried out from different directories. Otherwise, only the first use of *get* will succeed; since subsequent *get*s would attempt to over-write a *writable g-file*, they produce an SCCS error condition. In practice, this problem does not arise: normally such multiple executions are performed by different users[8] from different working directories.

Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved by *get*, as well as the SID of the version to be eventually created by *delta*, as a function of the SID specified to *get*.

[8] See the section entitled *Protection* for a discussion of how different users can use SCCS commands on the same files.

Table 1 — Determination of New SID

Case	*SID Specified**	*–b Option Used†*	*Other Conditions*	*SID Retrieved*	*SID of Delta to be Created*
1.	none‡	no	R defaults to mR	mR.mL	mR.(mL + 1)
2.	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB + 1).1
3.	R	no	R > mR	mR.mL	R.1§
4.	R	no	R = mR	mR.mL	mR.(mL + 1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
7.	R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB + 1).1
8.	R	–	Trunk successor in release > R and R exists	R.mL	R.mL.(mB + 1).1
9.	R.L	no	No trunk successor	R.L	R.(L + 1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB + 1).1
11.	R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mB + 1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS + 1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB + 1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S + 1)
15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB + 1).1
16.	R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB + 1).1

* 'R', 'L', 'B', and 'S' are the 'release', 'level', 'branch', and 'sequence' components of the SID, respectively; 'm' means 'maximum'. Thus, for example, 'R.mL' means 'the maximum level number within release R'; 'R.L.(mB + 1).1' means 'the first sequence number on the *new* branch (that is, maximum branch number plus 1) of level L within release R'. Note that if the SID specified is of the form 'R.L', 'R.L.B', or 'R.L.B.S', each of the specified components *must* exist.

† The –b option is effective only if the b flag (see *admin*(1)) is present in the file. In this table, an entry of '–' means 'irrelevant'.

‡ This case applies if the d (default SID) flag is *not* present in the file. If the d flag *is* present in the file, then the SID obtained from the d flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the *first* delta in a *new* release.

** 'hR' is the highest *existing* release that is lower than the specified, *nonexistent*, release.

18.1.5. Concurrent Edits of the Same SID

Normally, *get*s for editing (**-e** option specified) cannot operate concurrently on the same SID. Usually *delta* must be used before another **get -e** on the same SID. However, multiple concurrent edits (two or more *successive* **get -e** commands based on the same retrieved SID) *are* allowed if the **j** flag is set in the SCCS file. Thus:

```
tutorial% get -e s.abc
1.1
new delta 1.2
5 lines
tutorial%
```

may be immediately followed by:

```
tutorial% get -e s.abc
1.1
new delta 1.1.1.1
5 lines
tutorial%
```

without an intervening use of *delta*. In this case, a *delta* command corresponding to the first *get* produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the *delta* command corresponding to the second *get* produces delta 1.1.1.1.

18.1.6. Options That Affect Output

When the **-p** option is specified, *get* writes the retrieved text to the standard output, rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
tutorial% get -p s.abc > arbitrary-filename
```

The **-s** option suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, and so on, do not appear on the standard output. **-s** does not affect messages directed to the diagnostic output. **-s** is often used in conjunction with the **-p** option to 'pipe' the output of *get*, as in:

```
tutorial% get -p -s s.abc | nroff
```

A **get -g** verifies the existence of a particular SID in an SCCS but does not actually retrieve the text. This may be useful in a number of ways. For example,

```
tutorial% get -g -r4.3 s.abc
```

displays the specified SID if it exists in the SCCS file, and generates an error message if it doesn't. **-g** can also be used to regenerate a *p-file* that may have been accidentally destroyed:

```
tutorial% get -e -g s.abc
```

Get used with the **-l** option creates an *l-file*, which is named by replacing the 's.' of the SCCS file name with 'l.'. This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (format described in *get*(1)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
tutorial% get -r2.3 -l s.abc
```

generates an *l-file* showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of 'p' with the **-l** option, as in:

```
tutorial% get -lp -r2.3 s.abc
```

sends the generated output to the standard output rather than to the *l-file*. Note that the **-g** option may be used with the **-l** option to suppress the actual text-retrieval.

The **-m** option identifies the origin of each change applied to an SCCS file. **-m** tags each line of the generated *g-file* with the SID of the delta it came from. The SID precedes the line, and is separated from the text by a tab character.

When the **-n** option is specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab character. The **-n** option is most often used in a pipeline with *grep*(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory:

```
tutorial% get -p -n -s directory | grep pattern
```

If both the **-m** and **-n** options are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (the effect of the **-n** option), followed by the line in the format produced by the **-m** option.

Since using the **-m** option, the **-n** option, or both, modifies the contents of the *g-file*, such a *g-file* must *not* be used for creating a delta. Therefore, neither the **-m** nor the **-n** options may be used with the **-e** option.

See *get*(1) for a full description of additional *get* options.

18.2. delta — Make a Delta

Delta incorporates changes made to a *g-file* into the corresponding SCCS file. This process is known as 'making a delta', which is essentially a new version of the file.

Delta does a series of checks before creating the delta:

1. Searches the *p-file* for an entry containing the user's login name, because the user who retrieved the *g-file* must be the one who creates the delta. *Delta* displays an error message if the entry is not found. Note that if the login name of the user appears in more than one entry (that is, the same user did a **get -e** more than once on the same SCCS file), the **-r** option must be used with *delta* to specify an SID that uniquely identifies the *p-file* entry[9].
2. Performs the same permission checks as **get -e**.

If these checks succeed, *delta* compares the *g-file* (via *diff*(1)) with its own, temporary copy of the *g-file* as it was before editing, to determine what has been changed. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the 's.' of the SCCS file name with 'd.'); *delta* retrieves it by doing its own *get* at the SID specified in the *p-file* entry. If you would like to see the results of *delta*'s *diff*, use the **-p** option to display it on standard output.

In practice, the most common use of *delta* is:

```
tutorial% delta s.abc
```

If your standard output is a terminal, *delta* replies: 'comments?'. You may now type a response — usually a description of why the delta is being made — of up to 512 characters, terminating with a newline character. Newline characters *not* intended to terminate the response should be preceded by '\'.

If the SCCS file has a **v** flag, *delta* asks for 'MRs?' before prompting for 'comments?' (again, this prompt is printed only if the standard output is a terminal). Enter MR[10] numbers, separated

[9] The SID specified may be either the SID retrieved by *get*, or the SID *delta* is to create.

[10] In a tightly controlled environment, one would expect deltas to be created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and would think it desirable or necessary to record such MR number(s) within each

by blanks and/or tabs, and terminate your response with a newline character.

If you want to enter commentary (comments and/or MR numbers) directly on the command line, use the **-y** and/or **-m** options, respectively. For example:

```
tutorial% delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

inserts the 'descriptive comment' and the MR numbers 'mrnum1' and 'mrnum2' without prompting or reading from standard input. **-m** can only be used if the SCCS file has a **v** flag. These options are useful when *delta* is executed from within a *Shell procedure* (see *sh*(1)).

The commentary (comments and/or MR numbers), whether solicited by *delta* or supplied via options, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of *delta*. Thus if *delta* is used with more than one file argument, and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Only files conforming to these rules are processed.

After the prompts for commentary, and before any other output, *delta* displays:

```
No id keywords (cm7)
```

if it finds no ID keywords in the edited *g-file* while making a delta. If there *were* any ID keywords in the SCCS file, this might mean one of two things. The keywords may have been replaced by their values (if a *get* without the **-e** option was used to retrieve the *g-file*). Or, the keywords may have been accidentally deleted or changed while editing the *g-file*. Of course, the file may never have had any ID keywords. In any case, it is left up to you to decide whether any action is necessary, but the delta is made (unless there is an **i** flag in the SCCS file, which makes this a fatal error and kills the delta).

When processing is complete, *delta* displays a message containing the SID of the created delta (obtained from the *p-file* entry), and the counts of lines inserted, deleted, and left unchanged. Thus, a typical message might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

The reported counts may not agree with your sense of changes made; there are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and *delta* may describe the set differently than you. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*[11]. If there is only *one* entry in the *p-file*, the *p-file* itself is removed.

In addition, *delta* removes the edited *g-file*, unless the **-n** option is specified. Thus:

```
tutorial% delta -n s.abc
```

keeps the *g-file* upon completion of processing.

The **-s** (silent) option suppresses all output that is normally directed to the standard output, except the initial prompts for commentary. If you use **-s** with **-y** (and, possibly, **-m**), *delta* neither reads standard input nor writes to standard output.

---

delta.

[11] All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file* described in earlier.

18.3. admin — Administer SCCS Files

Admin administers SCCS files, that is, creates new SCCS files and changes parameters of existing ones. When an SCCS file is created, its parameters are either initialized by use of options or assigned default values if no options are supplied. The same options are used to change the parameters of existing files.

The two options used when detecting and correcting 'corrupted' SCCS files are discussed in the section entitled *Auditing*.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the *admin* command upon that file.

18.3.1. Creating SCCS Files

```
tutorial% admin -ifirst s.abc
```

creates the *initial* delta of the SCCS file 's.abc'. This delta contains the text from the file ('first') specified as the value of the –i option. If you use –i without a value, *admin* reads its text from standard input. Thus, the command:

```
tutorial% admin -i s.abc < first
```

produces the same result as our first example. If the text of the initial delta does not contain ID keywords, *admin* displays the warning message:

```
tutorial% admin -ifirst s.abc
No id keywords (cm7)
tutorial%
```

If you use the same *admin* command to set the **i** flag in the text (not to be confused with the –i option for *admin*), the message is treated as a fatal error and the SCCS file is not created. Only *one* SCCS file may be created at a time using the –i option.

When an SCCS file is created, the *release* number assigned to its first delta is normally '1', and its *level* number is always '1'. Thus, the first delta of an SCCS file is normally '1.1'. If you wish to specify a release number for the first delta, use the –r option:

```
tutorial% admin -ifirst -r3 s.abc
```

to name the first delta '3.1' rather than '1.1'. The –r option can only be used with the –i option, because –r is only meaningful in creating the first delta.

18.3.2. Inserting Commentary for the Initial Delta

You can use the –y and –m options with *admin*, just as with *delta*, to insert initial descriptive commentary and/or MR numbers when an SCCS file is created. If you don't use –y to comment, *admin* automatically inserts a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

If you want to supply MR numbers (–m option), the **v** flag must also be set (using the –f option described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a *delta commentary* in the SCCS file (see *sccsfile*(5)). Thus:

```
tutorial% admin -ifirst -mmrnum1 -fv s.abc
```

Note that the **–y** and **–m** options are only effective if a new SCCS file is being created.

18.3.3. Initializing and Modifiying SCCS File Parameters

The portion of the SCCS file reserved for ***descriptive text*** — see the section entitled *Descriptive Text* — may be initialized or changed through the use of the **–t** option. The descriptive text is intended as a summary of the contents and purpose of the SCCS file; actually its contents and length are up to you.

When an SCCS file is being created and the **–t** option is supplied, it must be followed by the name of a file from which the descriptive text should be taken. For example, the command:

```
tutorial% admin –ifirst –tdesc s.abc
```

specifies that the descriptive text is to be taken from file 'desc'.

When processing an *existing* SCCS file, the **–t** option specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
tutorial% admin –tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of 'desc'. Omitting the file name after the **–t** option *removes* the descriptive text from the SCCS file:

```
tutorial% admin –t s.abc
```

The ***flags*** — see the section entitled *Descriptive Text* — of an SCCS file may be initialized and changed with the **–f** (flag) option, or may be deleted with the **–d** (delete) option. The flags of an SCCS file direct certain actions of the various commands. See *admin*(1) for a description of all the flags. For example, the **i** flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The **–f** option sets a flag and, possibly, sets its value. For example:

```
tutorial% admin –ifirst –fi –fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value 'modname' specified for the **m** flag is the value that the *get* command uses to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the %M% ID keyword). Note that several **–f** options may be supplied on a single *admin* command, and that **–f** options may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **–d** option deletes a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
tutorial% admin –dm s.abc
```

removes the **m** flag from the SCCS file. Several **–d** options may be supplied on a single *admin* command, and may be interspersed with **–f** options.

SCCS files contain a list (*user list*) of login names and/or group IDs of users who are allowed to create deltas. This list is normally empty, implying that *anyone* may create deltas. To add login names and/or group IDs to the list, use the *admin* command with the **–a** option. For example:

```
tutorial% admin –awendy –aalison –a1234 s.abc
```

adds the login names 'wendy' and 'alison' and the group ID '1234' to the list. The **–a** option may be used whether *admin* is creating a new SCCS file or processing an existing one, and may appear several times. The **–e** option is used in an analogous manner if one wishes to remove ('erase') login names or group IDs from the list.

18.4. prs — Print SCCS File

Prs displays all or parts of an SCCS file on the standard output. The format of this display, called the output *data specification*, is set via the –d option.

The data specification is a string consisting of SCCS file *data keywords*[12] interspersed with (optional) text. Data keywords are replaced by appropriate values according to their definitions. For example: :I: is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, :F: is defined as the data keyword for the name of the file currently being processed, and :C: is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see *prs*(1).

There is no limit to the number of times a data keyword may appear in a data specification; *prs* will respond with as many substitutions as you call for:

```
tutorial% prs –d":I: this is the top delta for :F: :I:" s.abc
2.1 this is the top delta for s.abc 2.1
tutorial%
```

If you want *prs* to print from a single delta, use the –r option to specify the SID of that delta:

```
tutorial% prs –d":F: :I: comment line is: :C:" –r1.4 s.abc
s.abc: 1.4 comment line is: THIS IS A COMMENT
tutorial%
```

If the –r option is *not* specified, the value of the SID defaults to the most recently created delta.

If you want information from a *range* of deltas, use the –e or –l option. –e substitutes data keywords for the SID designated via the –r option and all deltas created *earlier*:

```
tutorial% prs –d:I: –r1.4 –e s.abc
1.4
1.3
1.2.1.1
1.2
1.1
tutorial%
```

–l substitutes data keywords for the SID designated via the –r option and all deltas created *later*:

```
tutorial% prs –d:I: –r1.4 –l s.abc
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
tutorial%
```

Prs substitutes data keywords for *all* deltas of the SCCS file if you use both the –e and –l options.

[12] Not to be confused with *get ID keywords*.

18.5. help — Ask for Help

Help displays explanations of SCCS commands, and of messages these commands may print. *Help* takes zero or more arguments, which are are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. *Help* has no concept of *options* or *file arguments*. If no argument is given, *help* prompts for one. When *help* cannot find any information on an argument, it displays an error message. Each argument is processed independently; an error resulting from one argument does *not* terminate processing of the others.

If an argument is a command, *help* 'explains' it by giving you its synopsis. For example, the following asks for help on the 'ge5' error message and information about the *rmdel* command:

```
tutorial% help ge5 rmdel
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
     rmdel -rSID name ...
tutorial%
```

18.6. rmdel — Remove a Delta

Rmdel removes a delta from an SCCS file — it should be reserved for cases in which incorrect, global changes were made to a delta.

The delta to be removed must be a 'leaf' delta; that is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, deltas 1.3.2.1 and 2.1 can be removed, and so on.

To remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either have created the delta being removed, or be the owner of the SCCS file and its directory.

You must specify the *complete* SID of the delta to be removed, preceded by **-r**. The SID must have two components for a trunk delta, and four components for a branch delta. Thus:

```
tutorial% rmdel -r2.3 s.abc
```

removes (trunk) delta '2.3' of the SCCS file.

Before removing the delta, *rmdel* checks the following things:

1. the *release* number (R) of the given SID satisfies the relation:

   $$\text{floor} \leq R \leq \text{ceiling}$$

2. the SID specified is *not* that of a version for which a *get* for editing has been executed and whose associated *delta* has not yet been made.
3. the login name or group ID of the user either appears in the file's *user list* or the *user list* is empty.
4. the release specified cannot be *locked* against editing (that is, if the l flag is set (see *admin*(1)), the release specified *must* not be contained in the list).

If these conditions are satisfied, the delta is removed. Otherwise, processing is terminated.

After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file is changed from 'D' (delta) to 'R' (removed).

18.7. cdc — Change Delta Commentary

Cdc changes the commentary supplied to a delta when it was created. *Cdc* has the same command syntax and restrictions as *rmdel*, but the delta to be processed does *not* have to be a leaf delta. For example:

```
tutorial% cdc -r3.4 s.abc
```

specifies that the commentary of delta '3.4' of the SCCS file is to be changed.

Cdc behaves like *delta* when it solicits *new* commentary. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (that is, superseded), and the new commentary is entered ahead of this comment line. The 'inserted' comment line records the login name of the user executing *cdc* and the time of its execution.

You can also use *cdc* to delete selected MR numbers associated with the specified delta by preceding them with '!'. For example, to insert 'mrnum3' and delete 'mrnum1' for delta 1.4:

```
tutorial% cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

18.8. what — Identify SCCS Files

What finds SCCS identifying information within *any* specified UNIX file. *What* does not use any options, nor does it treat directory names and a name of '-' (a lone minus sign) in any special way, as do other SCCS commands.

What searches the given file(s) for all occurrences of the string '@(#)', which is the replacement for the %Z% ID keyword (see *get*(1)). *What* then displays whatever follows that string until the first double quote ("), greater than (>), backslash (\), newline, or (non-printing) NUL character.

As an example, let's begin with the SCCS file 's.prog.c' (a C program), which contains the following line:

```
char id[] "%Z%%M%:%I%";
```

We then do the following *get*:

```
tutorial% get -r3.4 s.prog.c
```

and finally compile the resulting *g-file* to produce 'prog.o' and 'a.out'.
Using *what* as follows then displays:

```
tutorial% what prog.c prog.o a.out
prog.c:
        prog.c:3.4
prog.o:
        prog.c:3.4
a.out:
        prog.c:3.4
tutorial%
```

The string *what* searches for need not be inserted via an ID keyword of *get* — it may be inserted in any convenient manner.

18.9. sccsdiff — Compare Two Versions of an SCCS File

Sccsdiff compares two specified versions of one or more SCCS files, and displays the differences in *diff*-like format. The versions to be compared are specified with **-r**, as in *get*, and *must* be specified as the first two arguments to *sccsdiff* in the order in which they were created, that is, the older version is specified first. The **-p** option may be used after these two arguments to pipe the output of *sccsdiff* through *pr*(1). SCCS files to be processed are named last. *Sccsdiff* does *not* accept directory names or a name of '-' (a lone minus sign). An example:

```
tutorial% sccsdiff -r3.4 -r5.6 s.abc
```

18.10. comb — Combine Deltas

Comb generates a *Shell procedure* (see *sh*(1)) which tries to reconstruct new SCCS files leaner than their original counterparts. The generated Shell procedure is written on the standard output.

The rebuilding discards unwanted deltas and combines others. *Comb* is intended for those SCCS files with deltas so old that they are no longer useful; it should *only* be used a small number of times in the life of an SCCS file.

Used without options, *comb* preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the 'shape' of the SCCS file tree; 'middle' deltas on the trunk and on all branches of the tree are eliminated. In Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some options to *comb* are:

The **-p** option specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **-c** option specifies a *list* of deltas to be preserved. All other deltas are discarded. See *get*(1) for the syntax of such a list.

When used with the **-s** option, *comb* generates a Shell procedure, which, when run, produces *only* a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is a good idea to run *comb* with the **-s** option (in addition to any other desired options) *before* attempting any actual reconstructions.

Note that the Shell procedure which *comb* generates is *not* guaranteed to save any space — in fact, it is possible for the reconstructed file to be *larger* than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

18.11. val — Validate Characteristics of an SCCS File

Val determines if a file is an SCCS file meeting the characteristics specified by an optional list of options. Any characteristics not met are considered errors.

Val checks for the existence of a particular delta when the SID for that delta is *explicitly* specified via the **-r** option. The string following the **-y** or **-m** option is used to check the value set by the **t** or **m** flag respectively (see *admin*(1) for a description of the flags).

Val treats the special argument '-' (a lone minus sign) differently from other SCCS commands. When the - argument is specified, *val* reads the argument list from the standard input instead of from the command line. The standard input is read until end-of-file. Thus *val* can be used

once with different values for the option and file arguments. For example:

```
tutorial% val -
-yc -mabc s.abc
-mxyz -ypll s.xyz
^D
tutorial%
```

This sequence first checks if file 's.abc' has a value 'c' for its *type* flag and value 'abc' for the *module name* flag. Once processing of the first file is completed, *val* processes the remaining files, in this case 's.xyz', to determine if they meet the characteristics specified by the options associated with them.

Val returns an 8-bit code which is a disjunction of the possible errors detected — that is, each bit set indicates the occurrence of a specific error (see *val*(1) for a description of the possible errors and their codes). In addition, an appropriate diagnostic is printed unless suppressed by the **-s** (silent) option. A return code of '0' indicates all named files met the characteristics specified.

19. SCCS Files

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

19.1. Protection

SCCS relies on the capabilities of the UNIX operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (that is, changes made by non- SCCS commands). The only protection features provided directly by SCCS are the *release lock* flag, the *release floor* and *ceiling* flags, and the *user list.*

New SCCS files created by *admin* are given mode 444 (read only). It is best *not* to change this mode, as it prevents any direct modification of the files by non- SCCS commands. Further, directories containing SCCS files should be given mode 755, so that only the *owner* of the directory can modify its contents.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, for example, sub-systems of a large project.

SCCS files must have only *one* link (name). Commands that modify SCCS files do so by creating a temporary copy of the file (called the *x-file,* and, upon completion of processing, remove the old file and rename the *x-file.* If the old file has more than one link, removing it and renaming the *x-file* would break the link. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with 's.'.

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the 'owner' of the SCCS files and as the one who will 'administer' them (for example, by using *admin*). This user is termed the *.SM SCCS administrator* for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands

that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel* and *cdc* commands.

The interface program must be owned by the SCCS administrator, and must have the *set user ID on execution* bit on (see *chmod*(1)), so that the effective user ID is the administrator's user ID. This program's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the interface program for the duration of that command's execution. In this manner, the owner of an SCCS file can modify it at will. Other users whose *login* names or *group* IDs are in the *user list* for that file (but who are *not* its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *cdc*. The project-dependent interface program, as its name implies, must be custom-built for each project.

19.2. Layout of an SCCS File

SCCS files are composed of lines of ASCII text arranged in six parts, as follows:

Checksum	A line containing the 'logical' sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as its type, SID, date and time of creation, and commentary included.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in *sccsfile*(5). In the following, the *checksum* is the only portion of the file discussed.

Because SCCS files are ASCII files, they may be processed by various UNIX commands: editors such as *vi*(1), text processing programs such as *grep*(1), *awk*(1), and *cat*(1), and so on. This is quite useful when an SCCS file must be modified manually (for example, when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when one wants to simply 'look' at the file.

⇒ ***Extreme care should be exercised when modifying SCCS files with non-SCCS commands.***

19.3. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, all or part of an SCCS file is destroyed. SCCS commands (like most UNIX commands) display an error message when a file does not exist. In addition, SCCS commands use the *checksum* stored in the SCCS file to determine whether a file has been *corrupted* since it was last accessed (has lost data, or has been changed). The *only* SCCS command which will process a corrupted SCCS file is *admin* with the **-h** or **-z** options. This is discussed below.

SCCS files should be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to audit such files is to use *admin* with the **-h** option on them:

```
tutorial% admin -h s.file1 s.file2 ...
    or
tutorial% admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

corrupted file (co6)

is produced for that file. This process continues until all files have been examined. When examining directories (as in the second example above), the process just described does not detect *missing* files. A simple way to detect whether any files are missing from a directory is to periodically list the contents of the directory (using *ls*(1)), and compare the current listing with the previous one. Any file which appears on the previous list but not the current one has been removed by some means.

When a file has been corrupted, the method of restoration depends upon the extent of the corruption. If damage is extensive, the best solution is to restore the file from a backup copy. When damage is minor, repairing the file with your favorite text editor may be possible. If you do repair the file with the system's text processing capabilities, you must use *admin*(1) with the **-z** option to recompute the checksum to bring it into agreement with the actual contents of the file:

```
tutorial% admin -z s.file
```

After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

A *Debugging Tools* document
is currently in preparation.

Table of Contents

DC — An Interactive Desk Calculator

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 memory 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. *DC* 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.

1. 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:

1.1. 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 Bell Laboratories.

1.2. Binary Operators — + - * % ^

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.

1.3. s — Pop the Stack Into A Named Register

The **s***x* command pops the value from the top of the main stack and stores that value 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.

2. l — Push Contents of a Named Register Onto the Stack

The **l***x* command pushes the value in register *x* 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 **l** command and is treated as an error by the **L** command.

2.1. d — Duplicate the Top of Stack

The top value on the stack is duplicated.

2.2. p — Display the Value on the Top of Stack

The top value on the stack is printed. The top value remains unchanged.

2.3. f — Display All Register and Stack Values

All values on the stack and in registers are printed.

2.4. x — Execute the Top of Stack

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.

2.5. [...] — Put Character String on Top of Stack

puts the bracketed character string onto the top of the stack.

2.6. q — Quit From DC

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.

2.7. Comparison Operators — <x >x =x !<x !>x !=x

The top two elements of the stack are popped and compared. Register *z* is executed if they obey the stated relation. Exclamation point is negation.

2.8. v — Compute Square Root of Top of Stack

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.

2.9. ! — Execute a System Command

interprets the rest of the line as a UNIX command. Control returns to *DC* when the UNIX command terminates.

2.10. c — Clear the Stack

All values on the stack are popped; the stack becomes empty.

2.11. i — Use Top of Stack Value as Input Number Radix

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.

2.12. o — Use Top of Stack Value as Output Number Radix

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.

2.13. k — Use Top of Stack Value as a Scale Factor

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.

2.14. z — Push Value of Stack Level Onto Stack

The value of the stack level is pushed onto the stack.

2.15. ? — Execute a Line of Input from Input Source

A line of input is taken from the input source (usually the console) and executed.

3. Detailed Description

3.1. 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.

3.2. 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 memory 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.

3.3. 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.

3.4. 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.

3.5. 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.

3.6. 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.

3.7. 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.

3.8. 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

$$x_{n+1} = \quad (x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

3.9. 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.

3.10. Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The hexadecimal digits A–F correspond to the numbers 10–15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push the value of the input base on the stack.

3.11. 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.

3.12. 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.

3.13. 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.

3.14. 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 **s** 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.

3.15. 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.

3.16. 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
```

3.17. 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.

3.18. 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.

4. 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 (that is, 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.

5. References

L. L. Cherry, R. Morris, *BC - An Arbitrary Precision Desk-Calculator Language.*

K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

Table of Contents

BC — ARBITRARY PRECISION DESK-CALCULATOR

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX† time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- to do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

BC 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 Bell Laboratories.

1. 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
```

2. 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 (that is, 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 (that is,, 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.

3. 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.

4. 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.

5. 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.

6. 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
        }
}
```

7. 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.

8. 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.

9. Acknowledgement

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

10. 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*.

11. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [] is optional.

11.1. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

11.2. Comments

Comments are introduced by the characters /* and terminated by */.

11.3. 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.

11.4. Keywords

The following are reserved keywords:

ibase **if**
obase **break**
scale **define**
sqrt **auto**
length **return**
while **quit**
for

11.5. 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.

12. 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.

12.1. Primitive 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.

Simple identifiers are named expressions. They have an initial value of zero.

Array elements are named expressions. They have an initial value of zero.

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.

12.2. Function Calls

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.

sqrt

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.

length

length(*expression*)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale

scale(*expression*)

The result is the scale of the expression. The scale of the result is zero.

12.3. Constants

Constants are primitive expressions.

12.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

12.5. Unary operators

The unary operators bind right to left.

– expression

The result is the negative of the expression.

+ + named-expression

The named expression is incremented by one. The result is the value of the named expression after incrementing.

— named-expression

The named expression is decremented by one. The result is the value of the named expression after decrementing.

named-expression + +

The named expression is incremented by one. The result is the value of the named expression before incrementing.

named-expression —

The named expression is decremented by one. The result is the value of the named expression before decrementing.

12.6. Binary Operators

Exponentiation operator

The exponentiation operator binds right to left.

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:

$\min(a \times b, \max(\mathbf{scale}, a))$

Multiplicative operators

The operators *, /, % bind left to right.

*expression * expression*

The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, the scale of the result is:

$\min(a+b, \max(\textbf{scale}, a, b))$

expression / expression

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

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**

12.7. Additive operators

The additive operators bind left to right.

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.

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.

12.8. assignment operators

The assignment operators bind right to left.

named-expression = expression

This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression =+ expression
named-expression =- expression
named-expression = expression*
named-expression =/ expression
named-expression =% expression
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.

12.9. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*
expression != *expression*

13. 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.

14. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

14.1. 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.

14.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

14.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

14.4. If statements

if (*relation*) *statement*

The substatement is executed if the relation is true.

14.5. While statements

while (*relation*) *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

14.6. 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.

14.7. Break statements

break

break causes termination of a **for** or **while** statement.

14.8. 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.

14.9. 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.

14.10. 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.

14.11. 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.

Table of Contents

M4 — A MACRO PROCESSOR

M4 is a macro processor available on UNIX† 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.

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

† UNIX is a trademark of Bell Laboratories.

1. 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

2. 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).

3. 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)

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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
```

10. 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!

11. Summary of Built-ins

Each entry is preceded by the page number where it is described.

3 changequote(L, R)
1 define(name, replacement)
4 divert(number)
4 divnum
5 dnl
5 dumpdef(`name', `name', ...)
5 errprint(s, s, ...)
4 eval(numeric expression)
3 ifdef(`name', this if true, this if false)
5 ifelse(a, b, c, d)
4 include(file)
3 incr(number)
5 index(s1, s2)
5 len(string)
4 maketemp(...XXXXX...)
4 sinclude(file)
5 substr(string, position, number)
4 syscmd(s)
5 translit(str, from, to)
3 undefine(`name')
4 undivert(number,number,...)

12. 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.

13. References

[1] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Inc., 1976.

Table of Contents

LEX — A LEXICAL ANALYZER GENERATOR

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. Lex is designed to simplify interfacing with Yacc, described in the next chapter.

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.

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.

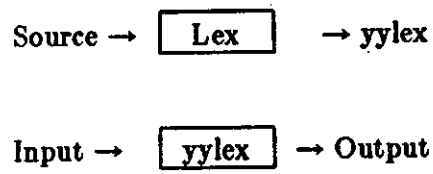


Figure 1: An overview of Lex

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 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.

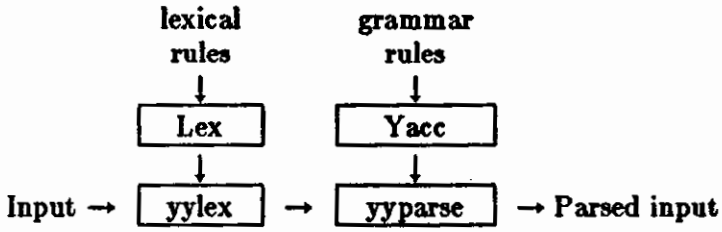


Figure 2: Lex with Yacc

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.

1. 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.

2. 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. For example, [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 *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by ***start conditions*** as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered 'being at the beginning of a line' to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named ***digit*** and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of ***a*** .

Finally, initial *%* is special, being the separator for Lex source segments.

3. Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]     ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "         |
"\t"        |
"\n"        ;
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a-z]+* . Lex leaves this text in an external character array named ***yytext***. Thus, to print the name found, a rule like

```
[a-z]+  printf("%s", yytext);
```

will print the string in ***yytext***. The C function ***printf*** accepts a format argument and data to be printed; in this case, the format is 'print string' (% indicating data conversion, and ***s*** indicating string type), and the data are the characters in ***yytext***.

So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+  ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches ***read*** it will normally match the instances of ***read*** contained in ***bread*** or

readjust ; to avoid this, a rule of the form *[a-z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+      {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*     {
        if (yytext[yyleng-1] == '\\')
            yymore();
        else
            ... normal user processing
        }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled 'normal processing'.

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of '=-a'. Suppose it is desired to treat this as '=- a' but print a message. A rule might be

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as '=-'. Alternatively it might be desired to treat this as '= -a'. To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of '=-3', however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) ***input()*** which returns the next input character;
2) ***output(c)*** which writes the character ***c*** on the output; and
3) ***unput(c)*** pushes the character ***c*** back onto the input stream to be read later by ***input()***.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. 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.

4. 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 (for example, ***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.

5. 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}         printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I* , which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under 'Summary of Source Format,' section 12.

6. 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 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.

7. 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.

8. 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     ;
%%
|s.
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';
                ECHO;
                }
```

To avoid such names as ***dsinz*** 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.

9. 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.

10. 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
```

Figure 3: 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.

11. 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

12. 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.

13. 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.

14. 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,
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,
6. D. M. Ritchie, private communication. See also M. E. Lesk, ***The Portable C Library,*** Computing Science Technical Report No. 31,

Table of Contents

YACC — YET ANOTHER COMPILER-COMPILER

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.

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[1] 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.[2, 3, 4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

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.[8] 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[2, 3, 4] 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.

9.1. 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.

9.2. 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!

NULL the attempt to open failed.

FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;

The stream named by **ioptr** is closed, if necessary, and then reopened as if by **fopen**. If the attempt to open fails, **NULL** is returned, otherwise **ioptr**, which will now refer to the new file. Often the reopened stream is **stdin** or **stdout**.

int getc(ioptr) FILE *ioptr;

returns the next character from the stream named by **ioptr**, which is a pointer to a file such as returned by **fopen**, or the name **stdin**. The integer **EOF** is returned on end-of-file or when an error occurs. The null character **\0** is a legal character.

int fgetc(ioptr) FILE *ioptr;

acts like **getc** but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

putc(c, ioptr) FILE *ioptr;

putc writes the character **c** on the output stream named by **ioptr**, which is a value returned from **fopen** or perhaps **stdout** or **stderr**. The character is returned as value, but **EOF** is returned on error.

fputc(c, ioptr) FILE *ioptr;

acts like **putc** but is a genuine function, not a macro.

fclose(ioptr) FILE *ioptr;

The file corresponding to **ioptr** is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. **fclose** is automatic on normal termination of the program.

fflush(ioptr) FILE *ioptr;

Any buffered information on the (output) stream named by **ioptr** is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, **stderr** always starts off unbuffered and remains so unless **setbuf** is used, or unless it is reopened.

exit(errcode);

terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls **fflush** for each output file. To terminate without flushing, use **_exit**.

feof(ioptr) FILE *ioptr;

returns non-zero when end-of-file has occurred on the specified input stream.

ferror(ioptr) FILE *ioptr;

returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar();

is identical to **getc(stdin)**.

putchar(c);

is identical to **putc(c, stdout)**.

char *fgets(s, n, ioptr) char *s; FILE *ioptr;

reads up to **n−1** characters from the stream **ioptr** into the character pointer **s**. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. **fgets** returns the first argument, or **NULL** if error or end-of-file

10. Advanced Topics

This section discusses a number of advanced features of Yacc.

10.1. 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.

10.2. 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.

10.3. 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*[5] 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.

11.1. References

1. B.W. Kernighan 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 6*(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. 18*(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 (December 1977). This paper is reprinted in this manual.
6. S.C. Johnson, 'A Portable Compiler: Theory and Practice,' *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B.W. Kernighan and L.L. Cherry, 'A System for Typesetting Mathematics,' *Comm. Assoc. Comp. Mach. 18* pp. 151-157 (March 1975). This paper is reprinted in the Sun *Editing and Text Processing* Manual.
8. M.E. Lesk, 'Lex — A Lexical Analyzer Generator,' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975). This paper is reprinted in this manual.

Appendix A. A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled 'a' through 'z', and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression 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 (for example, 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.

Table of Contents

ASSEMBLER REFERENCE MANUAL

This paper is the Programmer's Reference Manual for *as* – the assembler for the UNIX† system running on the Sun Workstation. *As* converts source programs written in *Assembler Language* into a form that the linker utility, *ld*(1) will turn into a program that is runnable on the UNIX operating system.

As provides the assembly language programmer with a minimal set of facilities to write programs in assembler language. Since the majority of programming is done in high level languages, *as* doesn't provide any elaborate macro facilities or conditional assembly features. It is assumed that the volume of assembly code produced is so small that these facilities aren't required.

This chapter describes the syntax and usage of the *as* assembler for the Motorola MC68000 microprocessor. The basic format of *as* is loosely based on the Digital Equipment Corp Macro-11 assembler described in DEC's publication DEC-11-0MACA-A-D but also contains elements of the UNIX PDP-11 *as*(1) assembler. The instruction mnemonics and effective address format are derived from a Motorola publication on the MC68000: the *MACSS MC68000 Design Specification Instruction Set Processor* dated June 30, 1979.

This is a *reference manual* as opposed to a treatise on writing in assembler language. It is assumed that the reader is familiar with the concepts of machine architecture, the reasons for an assembler, the ideas of instruction mnemonics, operands, and effective address modes, and assembler directives. It is also assumed that the reader is familiar with the MC68000 processor, its instruction set, its addressing modes, and especially the irregularities in them.

1. Overview and Layout of This Chapter

In this introduction, below, there is a short summary of how to use *as*, and its command line options. The rest of the chapter is organized into sections containing the following material in this order:

- Basic elements of an assembler language program.
- Description of the lexical elements (tokens) that make up an *as* program.
- Rules for expressions.
- Layout of an assembly language program — the rules for constructing statements, and the elements of each statement.

† UNIX is a trademark of Bell Laboratories.

- Discussion on the specifics of the MC68000 machine instructions, instruction mnemonics, addressing modes, and addressing categories.
- Assembler *directives* (pseudo-ops) that *as* supports. Assembler directives do not generate machine instructions, but instead they direct the assemblers actions, and do jobs such as reserving space, or generating initialized data.
- Error messages.
- Summary of the MC68000 machine instructions (opcodes), their layout, and the condition codes that are affected.
- New features of the MC68010 processor which provides virtual memory support.

2. How to Use the Assembler

This discussion assumes that the reader is using *as* on the UNIX operating system.

The assembler source code of the program should be in a file with a *.s* suffix. Suppose that your program is in a file called *parts.s*. To run the assembler, type the command:

```
% as parts.s
```

As runs silently (if there are no errors), and generates a file called *a.out*.

As also accepts several command line options. These are:

-o Place the output in the file specified by the name following the -o.

-R Make initialized data segments read only (actually the assembler places them at the end of the .text area).

-L Keep local (compiler generated) symbols that start with the letter L. This is a debugging feature. If the -L option is omitted, the assembler discards those symbols and does not include them in the symbol table.

-J Make all jumps to external symbols (jsr and jmp) PC relative rather than long absolute. This is intended for use when the programmer knows that the program is short. If there are any externals which are too far away, the loader will complain when the program is linked.

-d2 This is intended for small stand-alone programs. The assembler makes all program references PC relative and all data references short absolute. Note that the -J option does half this job anyway.

Readers should also consult the UNIX Programmer's Manual page for the *man* entry on *as*.

3. Notation

The notation used in this chapter is a somewhat modified Backus-Naur Form (BNF). A string of characters on its own stands for itself, for example:

```
WIDGET
```

is an occurrence of the literal string "WIDGET", and:

1983

is an occurrence of the literal constant 1983. An element enclosed in < and > signs is a non-terminal symbol, and must eventually be defined in terms of some other entities. For example,

<identifier>

stands for the syntactic construct called "identifier", which is eventually defined in terms of basic objects. A syntactic object followed by an ellipsis:

<thing>...

denotes one or more occurrences of *<thing>*. Syntactic objects which occur one after the other, as in:

first thing second thing

simply means an occurrence of *first thing* followed by *second thing*. Syntactic elements separated by a vertical bar sign (|), as in:

<letter> | *<digit>*

means an occurrence of *<letter>* or *<digit>* but not both. Brackets and braces define the order of interpretation. Brackets also indicate that the syntax described by the subexpression they enclose is optional. That is:

[*<thing>*]

denotes zero or one occurrences of *<thing>*, while:

{*<thing one>*|*<thing two>*}*<thing three>*

denotes a *<thing one>* or a *<thing two>*, followed by a *<thing three>*.

4. Further Reading

Motorola MC68000 16-bit Microprocessor User's Manual.

5. Elements of Assembly Language

This chapter covers the lexical elements which comprise an assembly language program. The next chapter discusses the rules for expressions and operand formation. Topics covered in this chapter are:

- *Character set* which the assembler recognizes,
- Rules for *identifiers*,
- Syntax for *numeric constants*,
- Syntax for *string constants*,
- Rules for *comments*,
- Layout of an assembler *source statement*.

An assembler language program is ultimately constructed from characters. Characters are combined to make up *lexical elements* or *tokens* of the language. Combinations of tokens then form assembler language *statements*, and sequences of statements then form an assembler program.

This section describes the basic lexical elements of *as*.

5.1. Character Set Which the Assembler Recognizes

As recognizes the following character set:

- The *letters* **A** through **Z** and **a** through **z**.
- The *digits* **0** through **9**.
- The ASCII *graphic characters* — the printing characters other than letters and digits.
- The ASCII *non-graphics*: space, tab, carriage-return, and newline (also known as line feed).

5.2. Identifiers

Identifiers are used to tag assembler statements (where they are called *labels*), as the location tag for data, and as the symbolic names of constants.

An identifier in an *as* program is a sequence of from 1 to 255 characters from the set:

- Upper case letters **A** through **Z**.
- Lower case letters **a** through **z**.
- Digits **0** through **9**.
- The characters underline (_), period (.), and dollar sign ($).

The first character of an identifier must not be numeric. Other than that restriction, there are a few other points to note:

- All 255 characters of an identifier are significant and are checked in comparisons with other identifiers.
- Upper case letters and lower case letters are considered distinct, so that **kit_of_parts** and **KIT_OF_PARTS** are two different identifiers.
- Although the period (.) and dollar sign ($) characters can be used to construct identifiers, they are reserved for special purposes (pseudo-ops for instance) and should not appear in user-defined identifiers.

Examples of Identifiers

Grab_Hold Widget Pot_of_Message MAXNAME

5.3. Numeric Labels

A numeric label consists of a digit 0 to 9 followed by a colon. As in the case of name labels, a numeric label assigns the current value of the location counter to the symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form

*n*b

refer to the first numeric label

n:

*b*ackwards from the reference;

*n*f

symbols refer to the first numeric label

n:

*f*orwards from the reference.

5.4. Local Labels

Local labels are a special form of identifier which are strictly local to a control section. Local labels provide a convenient means of generating labels for branch instructions and such. Use of local labels reduces the possibility of multiply-defined labels in a program, and separates entry point labels from local references, such as the top of a loop. Local labels cannot be referenced from outside of the current assembly unit. Local labels are of the form *n*$ where *n* is any integer. Valid local labels include:

1$ 27$ 394$

5.5. Scope of Labels

The *scope* of a label is the "distance" over which it is visible to other parts of the program which want to reference it. An ordinary label which tags a location in the program or data is visible only within the current assembly. An identifier which is designated as an external identifier via a **.globl** directive are visible to other assembly units at link time.

Local labels have a scope, or span of reference, which extends between one ordinary label and the next. Every time an ordinary label is encountered, all previous local labels associated with the current location counter are discarded, and a new local label scope is created. The following example illustrates the different scopes of the different kinds of labels:

```
first:    addl     d0,d1        | creates a new local label scope

100$:     addqw    #7,d3        | first appearance of 100$
          bccs     100$         | branches to the label above

second:   andl     #0x7ff,d4    | 100$ has gone away

100$:     cmpw     d1,d3        | this is a different 100$
          beqs     100$         | branches to the previous instruction

third:    movw     d0,d7        | now 100$ has gone away again
          beqs     100$         | generates an error message
```

The labels *first*, *second*, and *third* all have a scope which is the entire source file containing them. The first appearance of the local label 100$ has a scope which extends between *first* and *second*. The second appearance of the local label 100$ has a scope which extends between *second* and *third*. After the appearance of the label *third*, the branch to 100$ will generate an error message because that label is no longer defined in this scope.

5.6. Constants

There are two forms of constants available to *as* users, namely *numeric* constants and *string* constants. All constants are considered absolute quantities when they appear in an expression (see section 3 for a discussion on absolute and relocatable expressions).

5.7. Numeric Constants

As assumes that any token which starts with a digit is a numeric constant. *As* accepts numeric quantities in either decimal (base 10), hexadecimal (base 16), or octal (base 8) radices. Numeric constants can represent quantities up to 32 bits in length.

Decimal numbers consist of between one and ten decimal digits (0 through 9). The range of decimal numbers is between -2,147,483,648 and 2,147,483,647. Note that you can't have commas in decimal numbers even though they are shown here for readability. Note also that decimal numbers can't be written with leading zeros, because a number starting with a zero is taken as an octal constant, as described below.

Hexadecimal constants must start with the notation **0x** and can then have between one and eight hexadecimal digits. The hexadecimal digits consist of the decimal digits 0 through 9 and the hexadecimal digits **a** through **f** or **A** through **F**.IAs **s**

digit 0. There can then be from one to 11 octal digits (0 through 7) in the number. But note that 11 octal digits is 33 bits, so the largest octal number is 037777777777. The assembler generates an error message if the decimal digits 8 and 9 appear in octal constants.

5.8. String Constants

A string is a sequence of ASCII characters, enclosed in quote signs ".

Within string constants, the quote sign is represented by a backslash character followed by a quote sign. The backslash character itself is represented by two backslash characters. Any other character can be represented by a backslash character followed by one, two, or three octal digits. The table below shows the octal representation of some of the more common non printing characters.

Character	*Octal Representation*
Backspace	010
Horizontal Tab	011
Newline (Line-Feed)	012
Form-Feed	014
Carriage-Return	015

5.9. Assembly Location Counter

The assembly location counter is the period character (.). It is colloquially known as **dot**. When used in the operand field of any statement, **dot** represents the address of the first byte of the statement. Even in assembler directives, **dot** represents the address of the start of that assembler directive. For example, if **dot** appears as the third argument in a **.long** directive, the value placed at that location is the address of the first location of the directive — **dot** is not updated until the next machine instruction or assembler directive. For example:

```
Ralph:    movl .,a0  | load value of Ralph into a0
```

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generated code. However, the location where the code is stored may be changed by a direct assignment altering the location counter:

. = <*expression*>

This <expression> must not contain any forward references, and must not change from one pass to another. Storage area may also be reserved be advancing **dot**. For example, if the current value of **dot** is 1000, the direct assignment statement:

```
Table:    .=.+0x100
```

reserves 256 bytes (100 hexadecimal) of storage, with the address of the first byte as the value of Table. The next instruction is stored at address 1100. Also see the **.skip** assembler directive for another means of achieving the same effect.

The value of **dot** is always relative to the start of the current control section. For instance:

```
. = 0x1000
```

does not set **dot** to absolute location 0x1000, but to location 0x1000 relative to the start of the current control section. This practice is not recommended.

6. Expressions

Expressions are combinations of operands (numeric constants and identifiers) and operators, forming new values. The sections below define the operators which *as* provides, then gives the rules for combining terms into expressions.

6.1. Operators

Identifiers and numeric constants can be combined, via arithmetic operators, to form *expressions*. *As* provides *unary* operators and *binary* operators, described below.

Unary Operators

Operator	*Function*	*Description*
–	unary minus.	Performs a two's complement of its following argument.
~	logical negation	Performs a one's complement logical negation of its following argument.

Binary operators

Operator	*Function*	*Description*
+	Addition	Arithmetic addition of its arguments.
–	Subtraction	Arithmetic subtraction of its arguments.
*	Multiplication	Arithmetic multiplication of its arguments.
/	Division	Arithmetic division of its arguments. Note that division is *as* is *integer* division, which truncates towards zero.

Each operator is assumed to work on a 32 bit number. If the value of a particular term occupies only 8 bits or 16 bits, the short quantity is sign extended into a full 32-bit value.

6.2. Terms

A term is a component of an expression. A term may be one of the following:

- A numeric constant, whose 32-bit value is used. The assembly location counter, known as **dot**, is considered a number in this context.
- An identifier.
- An expression or term enclosed in parentheses **()**. Any quantity enclosed in parentheses is evaluated before the rest of the expression. This can be used to alter the normal left-to-right evaluation of expressions (for example, differentiating between a*b+ c and a*(b+ c)) or to apply a unary operator to an entire expression (for example, -(a*b+ c)).
- A term preceded by a unary operator. For example, both **double_plus_ungood** and **~double_plus_ungood** are terms.

  Multiple unary operators can be used in a term. For example, **--positive** has the same value as **positive**.

6.3. Expressions

Expression are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value.

If the operand only requires a single byte value, (a **.byte** directive or an **addq** instruction, for example), the low order eight bits of the expression are used.

If the operand only requires a single 16-bit word value, (a **.word** directive or an **movem** instruction, for example), the low order 16 bits of the expression are used.

Expressions are evaluated left to right with no operator precedence. Thus

```
1 + 2 * 3
```

evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied.

A missing expression or term is interpreted as having a value of zero. In this case, an *Invalid expression* error is generated.

An *Invalid Operator* error means that a valid end-of-line character or binary operator was not detected after the assembler processed a term. In particular, this error is generated if an expression contains a identifier with an illegal character, or if an incorrect comment character was used.

6.4. Absolute, Relocatable, and External Expressions

When an expression is evaluated, its value is either absolute, relocatable, or external:

An expression is absolute if its value is fixed.

- An expression whose terms are constants is absolute.
- An identifier whose value is a constant via a direct assignment statement is absolute.

- A relocatable expression minus a relocatable term is absolute, where both items belong to the same program section.

An expression is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked, or loaded into memory. All labels of a program defined in relocatable sections are relocatable terms.

Expressions which contain relocatable terms must only *add or subtract constants to their value.* For example, assuming the identifier *widget* was defined in a relocatable section of the program, then the following demonstrates the use of relocatable expressions:

widget	*is a simple relocatable term. Its value bears a constant relationship to the base address of the current control section.*
widget+ 5	*is a simple relocatable expression. Since the value of widget has a constant relationship to the base address of the current control section, adding a constant to it does not change its relocatable status.*
widget*2	*Not relocatable. Multiplying a relocatable term by a constant invalidates the relocatable status.*
2–widget	*Not relocatable, since the expression cannot be linked by adding widget's offset to it.*
widget–blivet	*Absolute, since the offsets added to widget and blivet cancel each other out.*

An expression is external (or global) if it contains an external identifier not defined in the current program. With one exception, the same restrictions on expressions containing relocatable identifiers apply to expressions containing external identifiers. The exception is that the expression

widget–blivet

is incorrect when both *blivet* is an external identifier. The reason is that you cannot subtract an external relocatable expression. In addition, you cannot multiply or divide *any* relocatable expression.

7. Layout of an Assembler Language Source Program

An *as* program consists of a series of *statements*. Each statement occupies exactly one line.

A *line* is a sequence of characters with a <newline> character at the end. Blank lines (which have only whitespace with a <newline> character at the end) are ignored. The maximum line length is 255 characters. Continuation lines are not supported in this assembler.

Multiple statements (see below) can appear on a line, separated by semicolon characters. But note that once a comment field of a statement has been started, a semicolon appearing in the comment is part of the comment, and not a statement separator.

The format of an *as* assembly language statement is:

[<*label field*>] <*op-code*> [<*operand field*>] [| <*comment*>]

It is possible to have an assembler language statement which consists of only a label field. It is also possible to have an assembler language statement which consists of only a comment. Then as a consequence of the above two statements, it is possible to have an assembler language statement which consists of just a label field followed by a comment field.

The fields of a statement can be separated by spaces (blanks) or tabs. There must be at least one space or tab separating the op-code field from the operand field, but spaces are unnecessary elsewhere because the label field is terminated by a colon and the comment field starts with a vertical bar. Spaces can also appear on either side of operators in operand field expressions. Spaces and tabs are significant when they appear in a character string (for instance, as the operand of an .ASCII pseudo-op) or in a character constant. In this case, a space or tab stands for itself.

7.1. Label Field

A *label* is an identifier which the programmer may use to tag the location of program and data objects. The format of a <*label field*> is:

<*identifier*>: [<*identifier*>:]...

If present, a label *always* occurs first in a statement and *must* be terminated by a colon:

```
sticky:                 | there is a label defined here.
```

More than one label may appear in the same source statement, each one being terminated by a colon:

```
presson: grab: hold:    | there are multiple labels defined here.
```

A maximum of 10 labels may be defined in a single source statement. The collection of label definitions in a statement is called the *label field.*

When a label is encountered in the program, the assembler assigns that label the value of the current location counter. The value of a label may be either absolute or relocatable. If the current value of the location counter is relocatable, the absolute value of the symbol is assigned when the program is linked via the UNIX system *ld*(1) command.

7.2. Operation Code Field

The operation code field of an assembly language statement identifies the statement as either a machine instruction or an assembler directive.

One or more spaces (or tabs) must separate the operation code field from the following operand field in a statement. Spaces or tabs are unnecessary between the label and operation code fields, but they are recommended to improve readability of the program.

A machine instruction is indicated by an instruction mnemonic. The assembly language statement is intended to produce a single executable machine instruction. The operation of each instruction is described in the manufacturer's user manual. Some conventions used in *as* for instruction mnemonics are described in section 4 and a complete list of the instructions is presented in the appendix.

An assembler directive, or pseudo-op, performs some function during the assembly process. It does not produce any executable code, but it may assign space in a program for data.

Note that *as* expects that all instruction mnemonics in the op-code field should be in *lower case only*. Use of any upper case letters in instruction mnemonic gives rise to an error message.

The names of register operands must also be in lower case only. This behavior differs from the case of identifiers, where upper case letters and lower case letters are considered distinct.

7.2.1. Operation Code Size Qualifiers

Many MC68000 machine instructions can operate upon byte (8-bit), word (16-bit), or long word (32-bit) data. The size which the programmer requires is indicated as part of the instruction mnemonic. For instance, a **movb** instruction moves a byte of data, a **movw** instruction moves a 16-bit word of data, and a **movl** instruction moves a 32-bit long word of data. In general, the default size for data manipulation instructions is word.

Similarly, branch instructions can use a long or short offset to indicate the destination. So the **beq** instruction uses a 16-bit offset, whereas the **beqs** uses a short (8-bit) offset.

Note that this implementation of *as* provides an extended set of branch instructions which start with the letter **j** instead of the letter **b**. If the programmer uses the **j** forms, the assembler computes the correct offset size for the instruction.

7.3. Operand Field

The *operand field* of an assembly language statement supplies the arguments to the machine instruction or assembler directive.

As makes a distinction between the <*operand field*> and individual <*operands*> in a machine instruction or assembler directive. Some machine instructions and assembler directives require two or more arguments, and each of these is referred to as an "operand".

In general, an operand field consists of zero or more operands, and in all cases, operands are separated by commas. In other words, the format for an <operand field> is:

[<*operand*>[,<*operand*>]...]

The format of the operand field for machine instruction statements is the same for all instructions, and is described in section 4. The format of the operand field for assembler directives depends on the directive itself, and is included in the directive's description in section 5 of this manual.

Depending upon the machine instruction or assembler directive, the *operand field* consists of one or more *operands*. The kinds of objects which can form an operand are:

- Register operands.
- Expressions.

These forms of operands are described in the subsections following.

7.3.1. Register Operands

Register operands in a machine instruction refer to the machine registers of the MC68000 processor. Register operands are:

- Any one of the *data* registers d0 through d7,
- Any one of the *address* registers a0 through a7,
- Any one of the *special* registers. The special registers are:
  - **cc** The Condition Code register.
  - **sr** The Status Register.
  - **sp** The Stack Pointer.
  - **usp** The User Stack Pointer.
  - **sfc** The Source Function Code register (68010 only).
  - **dfc** The Destination Function Code register (68010 only).

Note that register a7 and the stack pointer are the same register. The only place where this is important is when the supervisor must explicitly use **usp** to refer to the user stack pointer.

The notation d*n* refers to any data register, a*n* refers to any address register, and r*n* means any register from the data or address registers.

Note that register names *must* be in lower case; *as* does not recognize register names in upper case or a combination of upper case and lower case.

7.3.2. Operand Expressions

Expressions define rules for using arithmetic and logical *operators* to operate upon numeric constants and identifiers to yield new values. The rules for expressions were defined in chapter 3.

7.4. Comment Field

As provides the means for the programmer to place comments in the source code. There are two ways of representing comments:

A line whose first *non-whitespace* character is the octothorpe character (#) is considered a comment. This feature is handy for passing assembler code through the C preprocessor. For example, these lines are comments:

```
# This is a comment line.
    # And this one is also a comment line.
```

The other way to introduce a comment is when a comment field appears as a part of a statement. The comment field is indicated by the presence of the vertical bar character (|) after the rest of the source statement.

The comment field consists of all characters on a source line following and including the comment character. The assembler ignores the rest of the comment field up to the end of the line. Any character may appear in the comment field, with the obvious exception of the <newline> character, which starts a new line.

An assembler source line can consist of just the comment field. For example, the two statements below are quite acceptable to the assembler:

```
| This is a comment field.
| So is this.
```

7.5. Direct Assignment Statements

A direct assignment statement assigns the value of an arbitrary expression to a specified identifier. The format of a direct assignment statement is:

<identifier> = *<expression>*

Examples of direct assignments are:

```
vect_size   = 4
vectora     = 0xFFFE
vectorb     = vectora-vect_size
CRLF        = 0x0D0A
```

Any identifier defined by direct assignment may be redefined later in the program, in which case its value is the result of the last such statement. This is analogous to the SET operation found in other assemblers.

A local identifier may be defined by direct assignment, though this doesn't make much sense.

Register identifiers may not be redefined.

An identifier which has already been used as a label may not be redefined, since this would be tantamount to redefining the address of a place in the program. In addition, an identifier which has been defined in a direct assignment statement cannot later be used as a label. Both situations give rise to an assembler error message.

If the *<expression>* is absolute, the identifier is also absolute, and may be treated as a constant in subsequent expressions. If the *<expression>* is relocatable, however, the <identifier> is also relocatable, and it is considered to be declared the same program section as the expression.

If the *expression* contains an external identifier, the identifier defined by the = statement is also considered external. For example:

```
.globl X    | X is declared as external identifier
foo = X     | foo becomes an external identifier
```

assigns the value of X (zero if it is undefined) to foo and makes foo an external identifier. External identifiers may be defined by direct assignment.

8. Instructions and Addressing Modes

This chapter describes the conventions used in *as* to specify instruction mnemonics and addressing modes. The information in this chapter is specific to the machine instructions and addressing modes of the MC68000 processor.

8.1. Instruction Mnemonics

The instruction mnemonics which *as* uses are based on the mnemonics as described in the Motorola MC68000 processor manual. *As* deviates from the Motorola manual in several areas.

Most of the MC68000 instructions can apply to byte, word on long operands. Instead of using a qualifier of **.b**, **.w**, or **.l** to indicate byte. word, or long as in the Motorola assembler, *as* places a suffix after the normal instruction mnemonic, thereby creating a separate mnemonic to indicate which length operand was intended.

For example, there are three mnemonics for the *or* instruction: **orb**, **orw** and **orl**, meaning OR byte, OR word, and OR long, respectively.

Instruction mnemonics for instructions with unusual opcodes may have additional suffixes. Thus in addition to the normal *add* variations, there also exist **addqb**, **addqw** and **addql** for the *add quick* instruction.

Branch instructions come in two flavors, byte (or short) and word. uction appends the suffix *s* to the basic mnemonic to specify the short appends the suffix *s* to the basic mnemonic to specify the short version of the instruction. For example, **beq** refers to the word version of the Branch if Equal instruction, while **beqs** refers to the short version of that instruction.

8.2. Extended Branch Instruction Mnemonics

In addition to the instructions which explicitly specify the instruction length, *as* supports extended branch instructions, whose names are generally constructed by replacing the **b** with **j**.

If the operand of the extended branch instruction is a simple address in the current segment, and the offset to that address is sufficiently small, *as* automatically generates the corresponding short branch instruction.

If the offset is too large for a short branch, but small enough for a branch, the corresponding branch instruction is generated. If the operand references an external address or is complex (see next paragraph), the extended branch instruction is implemented either by a **jmp** or **jsr** (for **jra** or **jbsr**), or by a conditional branch (with the sense of the conditional inverted) around a

jmp for the extended conditional branches.

In this context, a complex address is either an address which specifies other than normal mode addressing, or relocatable expressions containing more than one relocatable symbol. For instance, if *a*, *b* and *c* are symbols in the current segment, the expression *a*+ *b*–*c* is relocatable, but not simple.

Consult appendix A for a complete list of the instruction op-codes.

8.3. Addressing Modes

The following table describes the addressing modes that *as* recognizes. The notations used in this table have these meanings:

a*n* refers to an address register,

d*n* refers to a data register,

r*i* refers to either a data or an address register,

d refers to a displacement, which is a constant expression in *as*,

xxx refers to a constant expression.

Certain instructions, particularly *move* accept a variety of special registers including:

sp the stack pointer which is equivalent to a7,

sr the status register,

cc the condition codes of the status register,

usp the user mode stack pointer,

pc the program counter.

Addressing Modes

Mode	*Notation*	*Example*
Register	a*n*,d*n*,sp,pc,cc,sr,usp	movw a3,d2
Register Deferred	a*n*@	movw a3@,d2
Postincrement	a*n*@+	movw a3@+ ,d2
Predecrement	a*n*@-	movw a3@-,d2
Displacement	a*n*@(*d*)	movw a3@(24),d2
Word Index	a*n*@(*d*, R*i*:W)	movw a3@(16, d2:W),d3
Long Index	a*n*@(*d*, R*i*:L)	movw a3@(16, d2:L),d3
Absolute Short	*xxx*:W	movw 14:W,d2
Absolute Long	*xxx*:L	movw 14:L,d2
PC Displacement	pc@(*d*)	movw pc@(20),d3
PC Word Index	pc@(*d*, R*i*:W)	movw pc@(14, d2:W),d3
PC Long Index	pc@(*d*, R*i*:L)	movw pc@(14, d2:L),d3
Normal	foo	movw foo,d3
Immediate	#xxx	movw #27+ 3,d3

Normal mode assembles as PC relative if the assembler can determine that this is appropriate, otherwise it assembles as absolute long

The notation for these addressing modes derived from the Motorola notation with the exception of the colon instead of period in index mode.

The Motorola manual presents different mnemonics (and in fact different forms of the actual machine instructions) for instructions that use the literal effective address as data instead of using the contents of the effective address. For instance, the Motorola manual uses the mnemonic **adda** for *add address*. *as* does not make these distinctions because it can determine the type of the operand from the form of the operand. Thus an instruction of the form:

```
avenue:  .word 0
    ...
    addl   #avenue,a0
```

assembles to the *add address* instruction because *as* can see that *avenue* is an address.

```
right_now:    =    40000
    ...
    adda   #right_now,a0
```

assembles to an *add immediate* instruction because *as* can see that *right_now* is a constant.

Because of this determination of operand forms, some of the mnemonics listed in the Motorola manual are missing mnemonics from the set of mnemonics that *as* recognizes.

The MC68000 is restrictive in that certain classes of instructions only accept limited subsets of the address modes above. For example, the *add address* instruction does not accept a data register as a destination.

as tries to check all these restrictions and generates the *illegal operand* error code for instructions that do not satisfy the address mode restrictions.

The next section below describes how the address modes are grouped into address categories.

8.4. Addressing Categories

The MC68000 groups the effective address modes into categories derived from the manner in which they are used to address operands. Note the distinction between address *modes* and address *categories*. There are 14 addressing *modes*, and they fall into one or more of four addressing *categories*. The addressing categories are defined here, followed by a table which summarizes the grouping of the addressing modes into the categories.

Data means that the effective address mode is used to refer to data operands such as a d register or immediate data.

Memory means that the effective address mode can refer to memory operands. Examples include all the a-register indirect address modes and all the absolute address modes.

Alterable means that the effective address mode refers to operands which are writeable (alterable). This category takes in every addressing mode except the PC-relative addressing modes and the immediate address mode.

Control means that the effective address mode refers to memory operands without any explicit size specification.

Some addressing categories can be combined. So the Motorola MC68000 manual mentions things like *Data Alterable Addressing Mode* to mean that the particular instruction can use the data addressing mode, or the alterable addressing mode, or either of those modes.

Table 1: Addressing Categories

Addressing Categories					
Addressing Mode	*Assembler Syntax*	*Data*	*Memory*	*Control*	*Alterable*
Register Direct	a*n*, d*n*, sp, pc, cc, sr, usp	X			X
A Register Indirect	a*n*@	X	X	X	X
A Register Indirect with Post Increment	a*n*@+	X	X		X
A Register Indirect with Pre Decrement	a*n*@-	X	X		X
A Register Indirect with Displacement	a*n*@(*d*)	X	X	X	X
A Register Indirect with Word Index	a*n*@(*d*,r*i*:W)	X	X	X	X
A Register Indirect with Long Index	a*n*@(*d*,r*i*:L)	X	X	X	X
Absolute Short	xxx:W	X	X	X	X
Absolute Long	xxx:L	X	X	X	X
PC Relative	pc@(*d*)	X	X	X	
PC Relative with Word Index	pc@(*d*,r*i*:W)	X	X	X	
PC Relative with Long Index	pc@(*d*,r*i*:L)	X	X	X	
Immediate Data	#*nnn*	X	X		

9. Assembler Directives

Assembler directives are also known as *pseudo operations* or *pseudo-ops*. Pseudo-ops are used to direct the actions of the assembler, and to achieve effects such as generating data. The following pseudo-ops are available in *as*:

Assembler Directives	
Pseudo Operation	*Description*
.ascii	Generates a sequence of ASCII characters.
.asciz	Generates a sequence of ASCII characters, terminated by a zero byte.
.byte	Generates a sequence of bytes in data storage.
.word	Generates a sequence of words in data storage.
.long	Generates a sequence of long words in data storage.
.text	Specifies that generated code be placed in the *text* control section until further notice.
.data	Specifies that generated code be placed in the *data* control section until further notice.
.data1	Specifies that generated code be placed in the *data1* control section until further notice.
.data2	Specifies that generated code be placed in the *data2* control section until further notice.
.bss	Specifies that space will be reserved in the *bss* control section until further notice.
.globl	Declares an identifier as global (external).
.comm	Declares the name and size of a *common* area.
.lcomm	reserves a specified amount of space in the *bss* area.
.skip	advances the location counter by a specified amount.
.even	forces location counter to next word (even byte) boundary.
.stab*x*	Builds special symbol table entries. These directives are here for the benefit of compilers which generate information for the symbolic debug utility.

These assembler directives are discussed in detail in the sections following.

9.1. .ascii — Generate Sequence of Character Data

The .ascii directive translates character strings into their ASCII equivalents for use in the source program. The format of the .ascii directive is:

[<*label*>:] **.ascii** "<*character string*>"

<*character string*>
contains any character or escape sequence which can appear in a character string. Obviously, a newline must not appear within the character string. A newline can be represented by the escape sequence \012.

The examples following illustrate the use of the **.ascii** statement:

Octal Code Generated:	*Statement:*
150 145 154 154 157 040 164 150 145 162 145	.ascii "hello there"
127 141 162 156 151 156 147 055 007 007 040 012	.ascii "Warning-\007\007 \012"
141 142 143 144 145 146 147	.ascii "abcdefg"

9.2. .asciz — Generate Zero Terminated Sequence of Character Data

The **.asciz** directive is equivalent to the **.ascis** directive with a zero byte automatically inserted as the final character of the string. This feature is indented for generating strings which C programs can use.
The examples following illustrate the use of the **.ascis** statement:

Octal Code Generated:	*Statement:*
110 145 154 154 157 040 127 157 162 144 041 000	.asciz "Hello World!"
124 150 105 040 107 162 145 141 164 040 120 122 117 115 160 153 151 156 040 163 164 162 151 153 145 163 040 141 147 141 151 156 041 000	.asciz "The Great PROMpkin strikes again!"

9.3. .byte, .word, .long — Generate Data

The **.byte**, **.word** and **.long** directives reserve bytes, words, and long words, and initializes them with specified values.
The format of the various forms of data generation statements is:

[<*label*>:] **.byte** [<*expression*>][,<*expression*>]...

[<*label*>:] **.word** [<*expression*>][,<*expression*>]...

[<*label*>:] **.long** [<*expression*>][,<*expression*>]...

The **.byte** directive reserves one byte (8 bits) for each expression in the operand field, and initializes the byte to the low-order 8 bits of the corresponding expression.

The **.word** directive reserves one word (16 bits) for each expression in the operand field, and initializes the word to the low-order 16 bits of the corresponding expression.

The **.long** directive reserves one long word (32 bits) for each expression in the operand field, and initializes the long word to the low-order 32 bits of the corresponding expression.

Multiple expressions can appear in the operand field of the **.byte**, **.word**, or **.long** directives. Multiple expressions must be separated by commas.

9.4. .text, .data, .bss — Switch Location Counter

These statements change the "control section" where assembled code will be loaded.

As (and the UNIX system linker) views programs as divided into three distinct sections or address spaces:

text is the address space where the executable machine instructions are placed.

data is the address space where initialized data is placed. The assembler actually knows about three data areas, namely, *data*, *data1*, and *data2*. The second and third data areas are mainly for the benefit of the C compiler and are of minimal interest to the assembly language programmer.

If the **-R** option is coded on the *as* command line, it means that the initialized data should be considered read only. It is actually placed at the end of the *text* area.

bss is the address space where the uninitialized data areas are placed. Also see the **.lcomm** directive described below.

For historical reasons, the different areas are frequently referred to as "control sections" (csects for short).

These sections are equivalent as far as *as* is concerned with the exception that no instructions or data are generated for the *bss* section — only its size is computed and its symbol values are output.

During the first pass of the assembly, *as* maintains a separate location counter for each section. Consider the following code fragments:

```
        .text     | place the next instruction in the text section
code:   movw       d1,d2

        .data     | now generate some data in the data section
grab:   .long      27

        .text     | now revert to the text section
more:   addw       d2,d1

        .data     | and now back to the data section
hold:   .byte      4
```

During the first pass, *as* creates the intermediate output in two separate chunks: one for the *text* section and one for the *data* section.

In the *text* section, *code* immediately precedes *more*; in the *data* section, *grab* immediately precedes *hold*. At the end of the first pass, *as* rearranges all the addresses so that the sections are sent to the output file in the order: *text*, *data* and *bss*.

The resulting output file is an executable image file with all addresses correctly resolved, with the exception of undefined *.globl's* and *.comm's*.

For more information on the format of the assembler's output file, consult the UNIX Programmers manual for the entry on *a.out*(5).

9.5. .skip — Advance the Location Counter

The **.skip** directive reserves storage area by advancing the current location counter a specified amount. The format of the **.skip** directive is:

```
.skip  <size>
```

where <size> is the number of bytes by which the location counter should be advanced. The **.skip** directive is equivalent to performing direct assignment on the location counter. For instance, a **.skip** directive like this:

```
.skip  1000
```

is equivalent to the direct assignment statement:

```
. = . + 1000
```

9.6. .lcomm — Reserve Space in .bss Area

The **.lcomm** directive is a lazy way to get a specific amount of space reserved in the *.bss* area. The format of the **.lcomm** directive is:

```
.lcomm  <name>,<size>
```

where <*name*> is the name of the area to reserve, and <*size*> is the number of bytes to reserve. The **.lcomm** directive specifically reserves the space in the *.bss* area, regardless of which location counter is currently in effect.

A **.lcomm** directive like this:

```
.lcomm    lower_forty,1200
```

is equivalent to these directives:

```
.bss          | switch to .bss area
lower_forty:  .skip  size
revert to previous control section
```

9.7. .globl — Designate an External Identifier

A program may be assembled in separate modules, and then linked together to form a single executable unit. See the *ld*(1) command in the UNIX Programmer's Manual.

External identifiers are defined in each of these separate modules. An identifier which is declared (given a value) in one module may be referenced in another module by declaring the identifiers as external in *both* modules.

There are two forms of external identifiers, namely, those defined with the **.globl** and those defined with the .comm directive. The **.comm** directive is described in the next section.

External symbols are declared with the **.globl** assembler directive. The format is:

.globl <*symbol*>[,<*symbol*>]...

For example, the following statements declare the array TABLE and the routine SRCH as external symbols:

```
        .globl    TABLE,SRCH
TABLE:  .word     0,0,0,0,0
SRCH:   movw      TABLE,d0
        etc...
```

External symbols are only *declared* to the assembler. They must be *defined* (that is, given a value) in some other statement by one of the methods mentioned above. They need not be defined in the current program; in this case they are flagged as "undefined" in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

9.8. .comm — Define the Name and Size of a Common Area

The **.comm** directive declares the name and size of common areas, for compatibility with FORTRAN and other languages which use common. The format of the **.comm** statement is:

.comm <*name*>, <*constant expression*>

where *name* is the name of the common area, and *constant expression* is the size of the common area.The **.comm** directive implicitly declares the identifier *name* as an external identifier.

as does **not** allocate storage for *common* symbols; this task is left to the linker. The linker computes the maximum declared size of each *common* symbol (which may appear in several load modules), allocates storage for it in the final *bss* section, and resolves linkages. If, however, <*name*> appears as a global symbol (label) in any module of the program, all references to <*name*> are linked to it, and no additional spaces is allocated in the *.bss* area.

9.9. .even — Force Location Counter to Even Byte Boundary

The **.even** directive advances the location counter to the next even byte boundary, if its current value is odd. This directive is necessary because word and long data values must lie on even byte boundaries, and also because machine instructions must start on even byte

boundaries.

10. Error Codes

Usage Errors

Unknown option 'x' ignored
As does not recognize the option *x*. Valid options are:

- -o Place the output in the file specified by the name following the -o.
- -R Make initialized data segments read only.
- -L Keep local (compiler generated) symbols that start with the letter L.
- -J Make all jumps to external symbols (jsr and jmp) PC relative rather than long absolute.
- -d2 Make all program references PC relative and all data references short absolute.

Cannot open source file
The assembler cannot open a specified source file. Check the spelling, ensure that the pathname supplied is correct, or check that you have read permission on that file.

Too many file names given
The assembler can't cope with the number of files given. Break the job into smaller stages.

Cannot open output file
The specified output file cannot be created. Check that the permissions allow opening this file.

Assembler Error Messages

If *as* detects any errors during the assembly process, it prints out a message of the form:

as: error (<line_no>): <error_code>

Error messages is sent to the Standard Error file. Here is a list of *as* error codes, and their possible causes.

Invalid Character
An unexpected character was encountered in the program text.

Multiply defined symbol

- An identifier appears twice as a label.
- An attempt to redefine a label using an = (direct assignment) statement.
- An attempt to use, as a label, an identifier which was previously defined in an = (direct assignment) statement.

Symbol storage exceeded
No more room is left in the assembler's symbol table. Cut the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Out of strings space
No more room is left in the assembler's internal string table. Cut the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Stab storage exceeded
No more room is left in the assembler's symbol table for debug information. Cut the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Invalid Constant
An invalid digit was encountered in a number. For example, using a decimal digit 8 or 9 in an octal number. Also happens when an out-of-range constant operand is found in an instruction – for example:

```
addq  #200,d0
asll  #12,d0
```

Invalid Term
The expression evaluator could not find a valid term: symbol, constant or [<*expression*>]. An invalid prefix to a number or a bad symbol name in an operand generates this message.

Invalid Operator
Check the operand field for a bad operator. The operators that *as* recognizes are plus (+), minus (-), negate or one's complement (~), multiply (*), and divide (/).

Non-relocatable expression

- If an expression contains a relocatable symbol (a label, for instance), the only operations that can be applied to it are the addition of absolute expressions or the subtraction of another relocatable symbol (which produces an absolute result).
- This message also appears when a reference is made to a local label which is undefined.

Invalid operand
The operand used is not consistent with the instruction used – for example:

```
addqb  #1,a5
```

is an invalid combination of instruction and operand. Check the instruction set descriptions for valid combinations of instructions and operands.

Invalid symbol
If an operand that should be a symbol is not – for example:

```
.globl 3
```

because the constant 3 is not a symbol.

Invalid assignment
An attempt was made to redefine a label with an == statement.

Too many labels
More than 10 labels appeared on a single statement.

Invalid op-code
The assembler did not recognize an instruction mnemonic. Probably a misspelling.

Invalid string
An invalid string was encountered in an **.ascii** or **.asciz** directive.

- Make sure the string is enclosed in double quotes.
- Remember that you must use the sequence \" to represent a double quote inside the string.

Wrong number of operands
Check the appendix containing the operation codes list for the correct number of operands for the current instruction.

Line too long
A statement was found which has more than 255 characters before the newline.

Invalid register expression
A register name was found where one should not appear – for example:

```
add  #d0,_there
```

Offset too large
The instruction is a relative addressing instruction and the displacement between this instruction and the label specified is too large for the address field of the instruction.

Odd address
The previous instruction or pseudo-op required an odd number of bytes and this instruction requires word alignment. This error can only follow an **.ascii**, an **.asciz**, or a **.byte** pseudo-operation.

- Use a **.even** directive to ensure that the location counter is forced to a 16-bit boundary.

Undefined L-symbol
This is a warning message. A symbol beginning with the letter "L" was used but not defined. It is treated as an external symbol. Compiler generated labels usually start with the letter "L" and should be defined in this assembly. The absence of such a definition usually indicates a compiler code generation error.

Appendix A. List of AS Opcodes

This appendix is a list of the instruction mnemonics, grouped into logical categories.

Each operation code describes the following things:

- The mnemonics for the instruction,
- The generic name for the instruction,
- The assembler syntax and the variations on the instruction,
- The condition codes that this instruction affects.

The syntax for *as* machine instructions differs somewhat from the instruction layouts and categories shown in the Motorola MC68000 manual. For example, *as* provides a single set of mnemonics for add (add binary), adda (add address), and addi (add immediate). In general, *as selects the appropriate instruction from the form of the operands.*

Here is a brief explanation of the notations used below.

- An instruction of the form add*x*, when describing the assembler syntax, means that the instruction is coded as addb or addw or addl, *etcetera.*
- An operand field of a*n* means any A register.
- An operand field of d*n* means any D register.
- An operand field of r*n* means any A or D register.
- An operand field of *ea* means an effective address designated by one of the permissible addressing modes for the MC68000. Consult the Motorola MC68000 manual for details of the allowed addressing modes for each instruction.
- An operand field of *#data* means an immediate operand.
- Other special registers such as cc (condition code register) and sr (status register) are specifically called out where appropriate.
- The condition code register has these flags, with the following meanings.
  - N Set if the most significant bit of the result is set. Cleared otherwise.
  - Z Set if the result is zero. Cleared otherwise.
  - V Set if there was an arithmetic overflow. Cleared otherwise.
  - C Set if a carry is generated (for addition) or a borrow is generated (for a subtraction) out of the most significant bit of the operand. Cleared otherwise.
  - X This condition code is transparent to data movement instructions. When it is affected it is set the same as the C (carry) condition.
- The notations under *condition codes* in the tables below have these meanings:
  - * set according to the result of the instruction.
  - \- this instruction does not affect this condition code.
  - 0 this instruction clears this condition code.
  - 1 this instruction sets this condition code.
  - U this condition code is undefined after the instruction.
  - ! this condition code is set according to the status register pulled off the stack, or according to the immediate operand.

Double Operand Instructions

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes* X	N	Z	V	C

addb
addw
addl
	add binary	addz *ea,dn*
		addz *dn,ea*
		addz *#data,ea*
			*	*	*	*	*

andb andw andl	logical and	andz *ea,dn* andz *dn,ea* andz *#data,ea*	*	*	*	*	*
cmpb cmpw cmpl	arithmetic compare	cmpz *ea,dn* cmpz *#data,ea*	-	*	*	*	*
eorb eorw eorl	logical exclusive or	eorz *dn,ea* eorz *#data,ea* eorb *#data,cc* eorw *#data,sr*	-	*	*	0	0
movb movw movl	move data	movz *ea,ea* movl *#data,dn*	-	*	*	0	0
orb orw orl	inclusive or	orz *ea,dn* orz *dn,ea* or *#data,ea* orb *#data,cc* orw *#data,sr*	-	*	*	0	0

subb
subw
subl
	arithmetic subtract	subz *ea,dn*
		subz *dn,ea*
		subz *#data,ea*
			*	*	*	*	*

Single Operand Instructions

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes* X	N	Z	V	C
clrb clrw clrl	clear an operand	clr*x* *ea*	-	0	1	0	0
negb negw negl	negate binary	neg*x* *ea*	*	*	*	*	*
negxb negxw negxl	negate binary with extend	negx*x* *ea*	*	*	*	*	*
notb notw notl	logical complement	not*x* *ea*	-	*	*	0	0
st	set all ones	st *ea*	-	-	-	-	-
sf	set all zeros	sf *ea*	-	-	-	-	-
shi	set high	shi *ea*	-	-	-	-	-
sls	set lower or same	sls *ea*	-	-	-	-	-
scc	set carry clear	scc *ea*	-	-	-	-	-
scs	set carry set	scs *ea*	-	-	-	-	-
sne	set not equal	sne *ea*	-	-	-	-	-
seq	set equal	seq *ea*	-	-	-	-	-
svc	set no overflow	svc *ea*	-	-	-	-	-
svs	set on overflow	svs *ea*	-	-	-	-	-
spl	set plus	spl *ea*	-	-	-	-	-
smi	set minus	smi *ea*	-	-	-	-	-
sge	set greater or equal	sge *ea*	-	-	-	-	-
slt	set less than	slt *ea*	-	-	-	-	-
sgt	set greater than	sgt *ea*	-	-	-	-	-
sle	set less than or equal	sle *ea*	-	-	-	-	-
tas	test operand then set	tas *ea*	-	*	*	0	0
tstb tstw tstl	test operand	tst*x* *ea*	-	*	*	0	0

Branch Instructions

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes* X	N	Z	V	C
bcc bccs	branch carry clear	bcc*z* *ea*	-	-	-	-	-
bcs bcss	branch carry set	bcs*z* *ea*	-	-	-	-	-
beq beqs	branch on equal	beq*z* *ea*	-	-	-	-	-
bge bges	branch greater or equal	bge*z* *ea*	-	-	-	-	-
bgt bgts	branch greater than	bgt*z* *ea*	-	-	-	-	-
bhi bhis	branch higher	bhi*z* *ea*	-	-	-	-	-
ble bles	branch less than or equal	ble*z* *ea*	-	-	-	-	-
bls blss	branch lower or same	bls*z* *ea*	-	-	-	-	-
blt blts	branch less than	blt*z* *ea*	-	-	-	-	-
bmi bmis	branch minus	bmi*z* *ea*	-	-	-	-	-
bne bnes	branch not equal	bne*z* *ea*	-	-	-	-	-
bpl bpls	branch positive	bpl*z* *ea*	-	-	-	-	-
bra bras	branch always	bra*z* *ea*	-	-	-	-	-
bsr bsrs	subroutine branch	bsr*z* *ea*	-	-	-	-	-
bvc bvcs	branch overflow clear	bvc*z* *ea*	-	-	-	-	-
bvs bvss	branch overflow set	bvs*z* *ea*	-	-	-	-	-

Test Conditions, Decrement and Branch

Mnemonics	Operation	Assembler Syntax	Condition Codes X	N	Z	V	C
dbcc	Decrement & Branch on Carry Clear	dbcc d*n,label*	-	-	-	-	-
dbcs	Decrement & Branch on Carry Set	dbcs d*n,label*	-	-	-	-	-
dbeq	Decrement & Branch on Equal	dbeq d*n,label*	-	-	-	-	-
dbf	Decrement & Branch on False	dbf d*n,label*	-	-	-	-	-
dbge	Decrement & Branch on Greater Than or Equal	dbge d*n,label*	-	-	-	-	-
dbgt	Decrement & Branch on Greater Than	dbgt d*n,label*	-	-	-	-	-
dbhi	Decrement & Branch on High	dbhi d*n,label*	-	-	-	-	-
dble	Decrement & Branch on Less Than or Equal	dble d*n,label*	-	-	-	-	-
dbls	Decrement & Branch on Low or Same	dbls d*n,label*	-	-	-	-	-
dblt	Decrement & Branch on Less Than	dblt d*n,label*	-	-	-	-	-
dbmi	Decrement & Branch on Minus	dbmi d*n,label*	-	-	-	-	-
dbne	Decrement & Branch on Not Equal	dbne d*n,label*	-	-	-	-	-
dbpl	Decrement & Branch on Plus	dbpl d*n,label*	-	-	-	-	-
dbra	Decrement & Branch Always (same as dbf)	dbra d*n,label*	-	-	-	-	-
dbt	Decrement & Branch on True	dbt d*n,label*	-	-	-	-	-
dbvc	Decrement & Branch on Overflow Clear	dbvc d*n,label*	-	-	-	-	-
dbvs	Decrement & Branch on Overflow Set	dbvs d*n,label*	-	-	-	-	-

Extended Branch Instructions

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes*				
			X	N	Z	V	C
jcc	jump carry clear	jcc *ea*	-	-	-	-	-
jcs	jump on carry	jcs *ea*	-	-	-	-	-
jeq	jump on equal	jeq *ea*	-	-	-	-	-
jge	jump greater or equal	jge *ea*	-	-	-	-	-
jgt	jump greater than	jgt *ea*	-	-	-	-	-
jhi	jump higher	jhi *ea*	-	-	-	-	-
jle	jump less than or equal	jle *ea*	-	-	-	-	-
jls	jump lower or same	jls *ea*	-	-	-	-	-
jlt	jump less than	jlt *ea*	-	-	-	-	-
jmi	jump minus	jmi *ea*	-	-	-	-	-
jne	jump not equal	jne *ea*	-	-	-	-	-
jpl	jump positive	jpl *ea*	-	-	-	-	-
jra	jump always	jra *ea*	-	-	-	-	-
jbsr	jump to subroutine	jbsr *ea*	-	-	-	-	-
jvc	jump no overflow	jvc *ea*	-	-	-	-	-
jvs	jump on overflow	jvs *ea*	-	-	-	-	-

Shift Instructions

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes* X	N	Z	V	C
aslb aslw asll	arithmetic shift left	aslz dz,dy aslz #data,dy aslz ea	*	*	*	*	*
asrb asrw asrl	arithmetic shift right	asrz dz,dy asrz #data,dy asrz ea	*	*	*	*	*
lslb lslw lsll	logical shift left	lslz dz,dy lslz #data,dy lslz ea	*	*	*	0	*
lsrb lsrw lsrl	logical shift right	lsrz dz,dy lsrz #data,dy lsrz ea	*	*	*	0	*
rolb rolw roll	rotate left	rolz dz,dy rolz #data,dy rolz ea	0	*	*	0	*
rorb rorw rorl	rotate right	rorz dz,dy rorz #data,dy rorz ea	0	*	*	0	*
roxlb roxlw roxll	rotate left with extend	roxlz dz,dy roxlz #data,dy roxlz ea	*	*	*	0	*
roxrb roxrw roxrl	rotate right with extend	roxrz dz,dy roxrz #data,dy roxrz ea	*	*	*	0	*

Miscellaneous Classes

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes* X	N	Z	V	C
abcd	add decimal with extend	abcd dy,dx abcd ay@-,ax@-	*	U	*	U	*
addqb addqw addql	add quick	addqz #data,ea	*	*	*	*	*
addxb addxw addxl	add extended	addxz dy,dx addxz ay@-,ax@-	*	*	*	*	*
bchg	test a bit and change	bchg dn,ea bchg #data,ea	-	-	*	-	-
bclr	test a bit and clear	bclr dn,ea bclr #data,ea	-	-	*	-	-
bset	test a bit and set	bset dn,ea bset #data,ea	-	-	*	-	-
btst	test a bit	btst dn,ea btst #data,ea	-	-	*	-	-
cmpmb cmpmw cmpml	compare memory	cmpmz ay@+ ,Ax@+	-	*	*	*	*
chk	check register against bounds	chk ea,dn	-	*	U	U	U
divs	signed divide	divs ea,dn	-	*	*	*	0
divu	unsigned divide	divs ea,dn	-	*	*	*	0
exg	exchange registers	exg rz,ry	-	-	-	-	-
extw extl	sign extend	ext dn	-	*	*	0	0
jmp	jump	jmp ea	-	-	-	-	-
jsr	jump to subroutine	jsr ea	-	-	-	-	-
lea	load effective address	lea ea,an	-	-	-	-	-
link	link and allocate	link an,#disp	-	-	-	-	-

Miscellaneous Classes, continued

Mnemonics	*Operation*	*Assembler Syntax*	*Condition Codes* X	N	Z	V	C
movem*l* movem*w*	move multiple registers	movem*z* *#mask,ea* movem*z* *ea,#mask*	-	-	-	-	-
movep*l* movep*w*	move peripheral	movep*z* d*n*,a*n*@(*d*) movep*z* d*n*,a*n*@(*d*)	-	-	-	-	-
moveq	move quick	moveq *#data*,d*n*	-	*	*	0	0
muls	signed multiply	muls *ea*,d*n*	0	0	*	*	0
mulu	unsigned multiply	mulu *ea*,d*n*	0	0	*	*	0
nbcd	negate decimal with extend	nbcd *ea*	*	U	*	U	*
nop	no operation	nop	-	-	-	-	-
pea	push effective address	pea *ea*	-	-	-	-	-
reset	reset machine	reset	-	-	-	-	-
rte	return from exception	rte	?	?	?	?	?
rtr	return and restore codes	rtr	?	?	?	?	?
rts	return from subroutine	rts	-	-	-	-	-
sbcd	subtract decimal with extend	sbcd d*y*,d*x* sbcd a*y*@-,a*x*@-	*	U	*	U	*
stop	halt machine	stop *#xxx*	?	?	?	?	?
subqb subqw subql	subtract quick	subq*z* *#data,ea*	*	*	*	*	*
subxb subxw subxl	subtract extended	subx*z* d*y*,d*x* subx*z* a*y*@-,a*x*@-	*	*	*	*	*
swap	swap register halves	swap d*n*	*	*	*	*	*
trap	trap	trap *#vector*	-	-	-	-	-
trapv	trap on overflow	trapv	-	-	-	-	-
unlk	unlink	unlk a*n*	-	-	-	-	-

B. MC68010 Extensions

The Motorola MC68010 processor has some additional instructions and some extensions to existing instructions. These are documented here. Here is a brief summary of the new features of the MC68010.

- When the processor takes a bus error or address error exception, it saves 29 words on top of the system stack. The software must be cognizant of whether to pop 29 words or four words on executing an RTE instruction.
- A Vector Base Register has been added so that the exception/trap vectors can be located anywhere in supervisor space. The only time that the startup vectors are read from absolute location 0 is on processor reset.

- Two new 3-bit registers have been added to provide supervisor access to alternate address spaces. The Source Function Code register (SFC) and the Destination Function Code register (DFC) are used in conjunction with the Move to Address Space (movs) instruction to control which address spaces is selected for the move.
- There is a new instruction — MOVC — which stands for MOVE to/from Control Register.
- A MOVE FROM CCR instruction has been added so that user programs can move the condition code register to a specified destination.
- The MOVE FROM STATUS REGISTER instruction is now privileged.
- A new instruction — MOVS — which stands for MOVE to/from Address Space has been added.
- The RTE instruction has been enhanced so that the instruction knows about different stack layouts, and by the addition of a field which gives control over the number of words added to the stack pointer when the RTE instruction is executed.
- The RTS instruction has been enhanced by the addition of a field specifying a number which should be added to the stack pointer after the program counter has been pulled off the stack. This means that a subroutine can automatically get its arguments popped off the stack when an RTS instruction is executed.

MOVC — Move To or From Control Register

The MOVC instruction moves data between an address or data register and the control register. The format of the MOVC instruction is:

```
MOVC   Rn,Cr
MOVC   Cr,Rn
```

The specified general register is copied to the specified control register, or *vice versa*. 32 bits are always transferred, even when the control register has fewer than 32 bits. Unused bits always read out as zeros.

MOVC is a privileged instruction.

MOVC does not affect any condition codes.

The layout of the MOVC instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
A/D	Register Number			Control Register											

The fields in the instruction have the following meanings:

dr field Specifies the direction of the data transfer:

0 Transfer is from C*r* to R*n*.

1 Transfer is from R*n* to C*r*.

A/D Field Specifies whether R*n* is an address or a data register:

0 R*n* is a data register.

1 R*n* is an address register.

Register Field

Specifies the number of the address or data register involved in the transfer.

Control Register

Specifies the number of the control register involved in the transfer. The only control register numbers defined are:

MC68010 Control Register Codes

Code	Name	Function
0x000	SFC	Source Function Code Register for the MOVS *ea* to R*n* instruction.
0x000	DFC	Destination Function Code Register for the MOVS R*n* to *ea* instruction
0x000	USP	User Stack Pointer.
0x000	VBR	Base Register for Exception Vector Table.

Any other numbers appearing in the control register field generate an illegal instruction exception.

Move From The Condition Code Register

This is a new instruction in the MC68010. The effect of moving the contents of the condition code register in the MC68000 was done via the Move from Status Register instruction. In the MC68010, the Move from Status Register instruction has been made privileged and the new Move from Condition Code Register added so that a user program can read the condition codes.

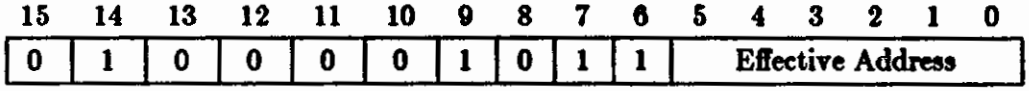
The format of the Move from Condition Code Register instruction is:

```
movw    cc,ea
```

A word composed of a high order byte of zeros, and the low order byte of the Status Register is written to the destination location specified by *ea*. This is a word only sized instruction.

Move from Condition Code Register does not affect any condition codes.

The layout of the Move from Condition Code Register instruction is:



The fields in the instruction have the following meanings:

Effective Address

Specifies the destination location where the Condition Code Register should be transferred. Only Data Alterable addressing modes are allowed in this instruction.

Move From The Status Register

The effect of moving the contents of the condition code register in the MC68000 was done via the Move from Status Register instruction. In the MC68010, the Move from Status Register instruction has been made privileged.

The format of the Move from Status Register instruction is:

```
movw    sr,ea
```

The contents of the Status Register is written to the destination location specified by *ea*. This is a word only sized instruction.

Move from Status Register is a privileged instruction.

Move from Status Register does not affect any condition codes.

The layout of the Move from Status Register instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Effective Address					

The fields in the instruction have the following meanings:

Effective Address

Specifies the destination location where the Status Register should be transferred. Only Data Alterable addressing modes are allowed in this instruction.

Move To/From Address Space

Move to or from Address Space moves a byte, word, or long word operand from a data or address register to an alternate address space, or moves a byte, word, or long word operand from a location in an alternate address space to a specified data or address register.

The format of the Move to/from Address Space instruction is:

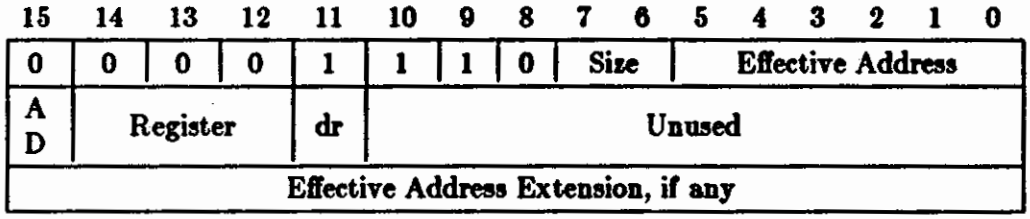
```
movsb   Rn,ea
movsw
movsl   ea,Rn
```

The address space involved in the transfer is determined by the Source Function Code (SFC) register (for a move from address space) and is determined by the Destination Function Code (DFC) register (for a move to address space).

Move to/from Address Space is a privileged instruction.

Move to/from Address Space does not affect any condition codes.

The layout of the Move to/from Address Space instruction is:



The fields in the instruction have the following meanings:

Size Field Specifies the size of the operation:

- 00 byte operation.
- 01 word operation.
- 10 long operation.

Effective Address

Specifies the source or destination location within the alternate address space. Only Memory Alterable addressing modes are allowed in this instruction.

Register Field

Specifies the number of the address or data register involved in the transfer.

A/D Field Specifies whether R*n* is an address or a data register:

- 0 R*n* is a data register.
- 1 R*n* is an address register.

dr field Specifies the direction of the data transfer:

- 0 Transfer is from effective address location within source address-space to R*n*.
- 1 Transfer is from R*n* to effective address location within destination address-space.

Return From Exception

Return from Exception is used when returning to a previous context after an interrupt or a trap has been processed.

The Status Register and the Program Counter are pulled from the system stack, and they overwrite the previous Status Register and Program Counter. The Vector Offset word is also pulled from the stack, and is examined to determine how much more information to restore.

The format of the Move to/from Address Space instruction is simply:

```
rte
```

The address space involved in the transfer is determined by the Source Function Code (SFC) register (for a move from address space) and is determined by the Destination Function Code (DFC) register (for a move to address space).

Move to/from Address Space is a privileged instruction.

All condition codes may be affected by the contents of the Status Register/Condition Code Register pulled off the stack.

The layout of the Move to/from Address Space instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

The Stack Format Field lies within the Vector Offset Word on the stack. The Vector Offset Word has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format				0	0	Vector Offset									

The Stack Format Field has the following meanings:

Value	Format	Description
0000	Short Format	Remove only four words from the top of the stack.
1000	Long Format	Remove 29 words from the top of the stack.
Anything Else	Bad Format	The processor takes a Stack Format Error Exception.

Return From Subroutine

Return from Subroutine is used when returning to a previous place in a program after executing the body of a subroutine. The MC68010 has enhanced the operation of the RTS instruction to specify a value to add to the stack pointer after the Program Counter is pulled from the stack.

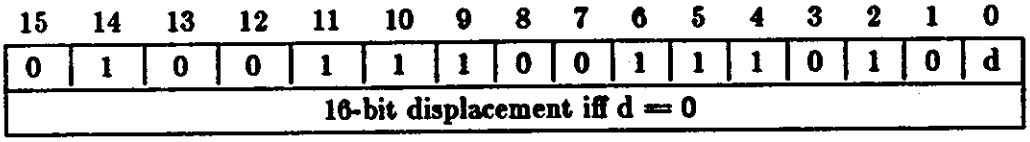
The Program Counter is pulled from the stack (either user or system), and it overwrites the previous Program Counter.

The format of the Return from Subroutine instruction is:

```
RTS
RTS   #n
```

The Return from Subroutine instruction does not affect any condition codes.

The layout of the Return from Subroutine instruction is:



The fields in the instruction have the following meanings:

d Field Specifies whether the displacement field should be added to the stack pointer:

0 16-bit sign-extended displacement is added to the stack pointer.

1 No displacement is added to the stack pointer.

Part Number 800-1112-01
Revision: C of 7 January 1984
For: Sun System Release 1.1

Editing and Text Processing

on the Sun Workstation

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

Credits and Acknowledgements

Material in this *Editing and Text Processing on the Sun Workstation* comes from a number of sources: *An Introduction to Display Editing with Vi*, William Joy, University of California, Berkeley, revised by Mark Horton; *Vi Command and Function Reference*, Alan P. W. Hewett, revised by Mark Horton; *Ex Reference Manual*, William Joy, revised by Mark Horton, University of California, Berkeley; *Awk — A Pattern Scanning and Processing Language*, Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, Bell Laboratories, Murray Hill, New Jersey; *Edit: A Tutorial*, Ricki Blau, James Joyce, University of California, Berkeley; *A Tutorial Introduction to the UNIX Text Editor*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Advanced Editing on UNIX*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Sed — a Non-Interactive Text Editor*, Lee. E. McMahon, Bell Laboratories, Murray Hill, New Jersey; *Nroff/Troff User's Manual*, Joseph F. Ossanna, Bell Laboratories, Murray Hill, New Jersey; *A Troff Tutorial*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *A Guide to Preparing Documents with -ms*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *Document Formatting on UNIX Using the -ms Macros*, Joel Kies, University of California, Berkeley, California; *Tbl — A Program to Format Tables*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *A System for Typesetting Mathematics*, Brian W. Kernighan, Lorinda L. Cherry, Bell Laboratories, Murray Hill, New Jersey; *Typesetting Mathematics — User's Guide*, Brian W. Kernighan, Lorinda L. Cherry, Bell Laboratories, Murray Hill, New Jersey; *Writing Tools — The Style and Diction Programs*, L. L. Cherry, W. Vesterman, Bell Laboratories, Murray Hill, New Jersey; *Updating Publications Lists*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *Some Applications of Inverted Indexes on the UNIX System*, M. E. Lesk, Bell Laboratories, Murray Hill, New Jersey; *Writing Papers with Nroff Using -me*, Eric P. Allman, University of California, Berkeley; and *-me Reference Manual*, Eric P. Allman, University of California, Berkeley. *Introducing the UNIX System*, Henry McGilton, Rachel Morgan, McGraw-Hill Book Company, 1983. These materials are gratefully acknowledged.

Revision History

Revision	Date	Comments
A	15 May 1983	First release of Editing and Text Processing.
B	1 November 1983	Updated and reorganized.
C	7 January 1984	New **-ms** macros; additions to document preparation introduction; and minor corrections.

Table of Contents

Preface

Editing and Text Processing on the Sun Workstation provides user's guides and reference information for the text editors and document processing tools. We assume you are familiar with a terminal keyboard and the Sun system. If you are not, see the *Beginner's Guide to the Sun Workstation* for information on the basics, like logging in and the Sun file system. If you are not familiar with a text editor or document processor in general, read *An Introduction to Text Editing* and *An Introduction to Document Preparation* in this manual for descriptions of the basic concepts and some simple examples that you can try. Finally, we assume that you are using a Sun Workstation, although specific terminal information is also provided.

If you choose to read one of the user's guides, sit down at your workstation and try the exercises and examples. The reference sections provide additional explanations and examples on how to use certain facilities and can be dipped into as necessary. For additional details on Sun system commands and programs, see the *User's Manual for the Sun Workstation*.

Use the table of contents to Part One and Part Two as roadmaps to guide you to the information you need.

Part One of this manual provides information on the text editors and Part Two describes the document formatting tools.

The contents of Part One are:

1. *An Introduction to Text Editing* — Describes the basics of text editing and provides a guide to the available editing tools. Newcomers should start here.
2. *Using 'vi', the Visual Display Editor* — Tutorial and reference information on the visual display editor *vi*. Includes a quick reference to tape up by your workstation.
3. *Command Reference for the 'ex' Line Editor* — A command reference for the *ex* and *vi* editors. Also includes a quick reference.
4. *Using the 'ed' Line Editor* — Provides a user's guide to the *ed* tools.
5. *Using 'sed', the Stream Text Editor* — A user's guide to *sed*, the non-interactive variant of *ed* for processing large files.
6. *Pattern Scanning and Processing with 'awk'* — A user's guide to the *awk* programming language for data transformation and selection operations.

Part Two contains the following chapters:

1. *An Introduction to Document Preparation* — Describes the basics of text processing, macros and macro packages, provides a guide to the available tools and several simple examples after which to pattern your papers and documents. Newcomers to the Sun document formatters should start here.
2. *Formatting Documents with the -ms Macros* — User's guide and reference information for the **–ms** macros for formatting papers and documents. Includes new **–ms** macros.

3. *Formatting Documents with 'nroff' and 'troff'* — Provides a user's guide and reference material for the *nroff* and *troff* text processors.
4. *Formatting Tables with 'tbl'* — A user's guide and numerous examples to the table processing utility *tbl*.
5. *Typesetting Mathematics with 'eqn'* — A user's guide to the *eqn* mathematical equation processor.
6. *Making Bibliographic References with 'refer'* — Explains how to use the bibliographic citation program *refer*.
7. *Formatting Documents with the -me Macros* — Describes the **-me** macro package for producing papers and documents.

Throughout this manual we use 'logo%' as the hostname to which you type system commands. **Bold face type** indicates commands that you type in as is. *Italics* specifies Sun system command names, general arguments or parameters that you should replace with a specific word or string, and important terms.

Table of Contents

List of Tables

Chapter 1

An Introduction to Text Editing

An *editor* is a utility program that you use to modify the contents of a file. A *text* editor deals with files containing a *string* of characters in a particular character set. A *string* is a sequence of characters, 'ABC,' 'evan' or 'm3154' for example. You usually use an editor *interactively*; that is, you can see on the workstation screen what you have and then make changes accordingly.

With a text editor, you can browse through a file, make changes, and then make the changes permanent.

There are also utilities such as *awk*, *grep*, *fgrep*, *egrep*, and *tr* that operate on a file, but do not change the original file. Rather they modify the data contained in it as the data goes from the original file to the workstation screen, printer, or whatever. Moreover, these commands operate on a global basis, that is, they change everything that conforms to a specific regular pattern. See *Pattern Scanning and Processing with 'awk'* in this manual for more information and the *User's Manual for the Sun Workstation* for details on the other utilities.

There are two kinds of editors, *line* editors and *screen* editors. A *line* editor has a line as its basic unit for change. A line is a string of characters terminated by a newline character, the character that is generated when you type RETURN. You can give the editor commands to do operations on lines, display, change, delete, move, copy a line, or insert a new line. You can substitute character strings within a line or group of lines.

A *screen* editor displays a portion of a file on the workstation screen. You can move the cursor around the screen to indicate where you want to make changes, and you can choose which part of the file to display. Screen editors, such as *vi*, are also called *display* editors.

1.1. Sun System Editors

The Sun system has two basic editors. *Ed* is the basic, interactive line editor from which the others have been developed. As they are all related, you can see similarities with *vi*, *ex*, and *sed*. Your primary interface to the Sun system is probably *vi* for editing both source code and text. See *Using the 'ed' Line Editor* for details on *ed*.

The other basic editor is the *stream* editor *sed*, which as a lineal descendant of *ed*, can perform similar operations. However, it is not interactive and you cannot move backwards in the edit file. You specify the command or series of commands to be executed, and *sed* performs them from the beginning to the end of the file. Because *sed* does not copy your file into the buffer to create a temporary file like *ed* does, you can use *sed* to edit any size file. *Sed* is usually used for making transient changes only. *Sed* recognizes basically the same *regular expressions* as *ed*. Regular expressions are described below. See *Using 'sed', the Stream Text Editor* for instructions on how to use *sed*.

More useful for general text editing are the screen editors *ex* and *vi*. A variant of *ex*, *edit*, has features designed to make it less complicated to learn and use.[1]

[1] See *Edit: A Tutorial*, Ricki Blau and James Joyce, University of California, Berkeley.

Ex is also based on *ed*, but has many extensions and additional features. Commands are less cryptic and hence, easier to remember. There are variants of some editor operations, which modify the way in which those operations are performed under certain conditions. *Ex* is more communicative, displaying more descriptive error messages than merely '?' as *ed* does and providing instructions on how to override the error condition. There are editor *options* which modify overall *ex* behavior. *Ex* also provides the *visual* mode, which turns *ex* into a screen editor. In this mode, *ex* is identical to *vi*. You can use the *open* mode for intraline editing.

Vi is the *screen*, *display* or *visual* editor version of *ex*. A portion of the file you wish to modify is displayed on your workstation screen. Within the displayed portion of the file, you can move the cursor around to control where changes are to be made, and then you can make changes by replacing, adding or deleting text. You can change the portion of the file displayed on the screen, so you have access to the whole file.

You also have access to all of the *ex* line-oriented commands from *vi*. Many of the more useful operations that can be performed in *vi* simply call upon *ex* functions. Additionally, some operations, such as global substitutions, are easily performed using *ex* from *vi*. Because of this connection, refer to both *Using 'vi,' the Visual Display Editor* and the *Command Reference for the 'ex' Line Editor*. For a quick tutorial on the most useful *vi* commands and features, read the chapter on *vi* in the *Beginner's Guide to the Sun Workstation*.

1.2. Text Editing Basics

In editing jargon, we say you *enter* an editor to edit a file and *quit* an editor to return to the system command level Shell.

Most editors set aside a temporary work space, called a *buffer*, separate from your permanent file. Before starting to work on an existing file, the editor makes a copy of it in the buffer, leaving the original untouched. When you make editing changes to the buffer copy, you must then save or *write* the file to make the changes permanent. The buffer disappears at the end of the editing session.

During an editing session there are two usual modes of operation: *command* mode and *text input* mode. (This disregards, for the moment, *open* and *visual* modes, discussed below.) In command mode, the editor may prompt you with '?,' a colon (:), or nothing at all as in *vi*. In text input mode, there is no prompt and the editor merely adds the text you type in to the buffer. You start text input mode with a command that *appends*, *inserts*, or *changes*, and terminate it either by typing a period as the first and only character on a line for *ed* and *ex* or by typing the ESCAPE (ESC) key for *vi*.

The editor keeps track of lines of text in the buffer by numbering them consecutively starting with 1 and renumbering as lines are added or deleted. It doesn't normally display the line numbers, although you can specify that they be displayed in *vi*. At any given time the editor is positioned at one of these lines; this position is called the *current line*.

Some editor commands take line-number prefixes. The concept of line numbers is especially important in *ed* and *ex*; you use them to indicate which lines to operate on. You also use line numbers in *vi*, but less frequently. With *ed*, you can precede most commands by one or two line-number addresses which indicate the lines to be affected. If you give one line number, the command operates on that line only; if you give two, it operates on an inclusive range of lines. Commands that can take line-number prefixes also assume default prefixes if none are given. The default assumed by each command is designed to make it convenient to use in many instances without any line-number prefix. For the most part, a command used without a prefix operates on the current line, though there are exceptions to this rule. The *print* command by

itself, for instance, displays one line, the current line, on the workstation screen. In the address prefix notation, '.' stands for the current line and '$' stands for the last line of the buffer. If no such notation appears, no line-number prefix may be used. Some commands take trailing information.

Besides command and text input modes, *ex*, *vi* and *edit* provide other modes of editing called *open* and *visual*. In these modes you can move the cursor to individual words or characters in a line. The commands you then use are very different from the standard editor commands; most do not appear on the screen when typed.

1.2.1. Regular Expressions in Text Patterns

You can use the editors and the utilities mentioned above to deal with fixed strings of characters, but this may become tedious if you want to do something more complex. You can also specify a pattern or template of text you want to modify; this pattern is called a *regular expression*. Certain characters take on special meanings when used in in these patterns. These special characters are called *metacharacters* because they represent something other than themselves.

Here is a table of all the special metacharacters and which utilities support those particular characters.

Table 1-1: Utilities and Their Metacharacters

Character	Meaning	Supported by *fgrep*	*grep*	*egrep*	*awk*	*ed*	*sed*	*ez*
c	any character except specials	yes	yes	yes	yes	yes	yes	yes
^	match beginning of line	yes	yes	yes	yes	yes	yes	yes
$	match end of line	yes	yes	yes	yes	yes	yes	yes
\<	match beginning of word	no	yes	no	no	no	no	yes
\>	match end of word	no	yes	no	no	no	no	yes
.	any character	no	yes	yes	yes	yes	yes	yes
[*string*]	character class	no	yes	yes	yes	yes	yes	yes
[^ *string*]	negated character class	no	yes	yes	yes	yes	yes	yes
*	closure	no	yes	yes	yes	yes	yes	yes
(*pattern*)	grouping	no	no	yes	yes	no	no	no
|	alternation	no	no	yes	yes	no	no	no
\(*pattern*\)	remember *pattern*	no	no	no	no	yes	yes	yes

To use one of these special characters as a simple graphic representation rather than its special meaning, precede it by a backslash ('\'). The backslash always has this special *escape* meaning.

Some of the metacharacters that *ed* and some of the other utilities use are also used by the Shell for matching filenames, so you should enclose the regular expression in single quotes (').

You can combine regular expressions to specify a lot more than just a single string of text, so you can give the editor commands that operate on either a very specific string of text or *globally* on a whole file.

See the *Beginner's Guide to the Sun Workstation* for a more detailed and descriptive explanation of regular expressions.

1.3. What to Do If Something Goes Wrong

Sometimes you may make a mistake or your system may not respond correctly. Here are some suggestions on what to do.

If you make a mistake in the editor that you cannot recover from easily, do not panic. As long as you do not *write* the file and quit the editor, you can retrieve the original file. Force the editor to quit (in *vi*, for example, you type **:q!**, the exclamation point overriding any warning), and then enter the editor again to start over. When you try to quit the editor without saving changes, the editor will warn you that you have unsaved changes, so you have to force the quit with '!'.

At the Sun system level, if you make a typing mistake, and see it before you press RETURN, there are several ways to recover. The DEL key is the *erase* character. Use it to back up over and *erase* the previously typed character. Successive uses of DEL erase characters back to the beginning of the line, but not beyond. Use **^C**[2] to abort or *send an interrupt* to a currently running program. You can't interrupt an editor with ^C.

Sometimes you can get into a state where your workstation or terminal acts strangely. For example, you may not be able to move the cursor, your cursor may disappear, there is no echoing of what you type, or typing RETURN may not cause a linefeed or return the cursor to the left margin. Try the following solutions:

- First, type **^Q** to resume possibly suspended output. (You might have typed **^S**, freezing the screen.)
- Another possibility is that you accidentally typed a NO SCRL key (also called SET UP/NO SCROLL on some terminals) on your keyboard. This also freezes the keyboard like typing a **^S**. Try typing **^Q**, which toggles you back to proper operation if you did indeed type the NO SCRL key in the first place.
- Next, try pressing the LINEFEED key, followed by typing 'reset', and pressing LINEFEED again.
- If that doesn't help, try logging out and logging back in. If you are using a terminal, try powering it off and on to regain normal operation.
- If you get unwanted messages or garbage on your screen, type **^L** to refresh the workstation screen. (Use **^R** on a terminal.)

If your system goes down, a file with almost all your latest changes is automatically saved. After rebooting your system, or doing whatever needs to be done, you will receive mail indicating that the file has been saved. First, return to the directory where the file belongs, and then re-enter the editor with the **–r** option to *restore* the file:

logo% **vi –r** *filename*

This returns you to a version of the file you were editing, minus a few of your most recent changes.

[2] We use the convention '^ *whatever*' to mean control-whatever — that is, hold down the control (or CTRL) key while typing a *whatever* character. '^C' means hold down the CONTROL key while typing 'c'. The case does not matter; ^C and ^c are equivalent.

Table of Contents

List of Tables

Chapter 2

Using vi, the Visual Display Editor

This chapter[1] describes *vi* (pronounced *vee-eye*) the visual, display editor. The first part of this chapter provides the basics of using *vi*. The second part provides a command reference and terminal set-up information. Finally, there is a quick reference, which summarizes the *vi* commands. Keep this reference handy while you are learning *vi*. As the *vi* editor is the visual display version of the *ex* line editor, and because the full command set of the line-oriented *ex* editor is available within *vi*, you can use the *ex* commands in *vi*. Some editing, such as global substitution, is more easily done with *ex*. So refer to the information in the *Command Reference for the 'ex' Line Editor* as it also applies to *vi*.

This chapter assumes you are using *vi* on the Sun Workstation. If you are using *vi* on a terminal, refer to *Terminal Information* for instructions on setting up your terminal.

In the examples, input that must be typed as is will be presented in **bold face**. Text which you should replace with appropriate input is given in *italics*.

2.1. Vi and Ex

As noted above, *vi* is actually one mode of editing within the editor *ex*. When you are running *vi* you can escape to the line-oriented editor *ex* by typing **Q**. All of the : commands which are introduced in *File Manipulation Commands* are available in *ex*. This places the cursor on the command line at the bottom of the screen. Likewise, most *ex* commands can be invoked from *vi* using :. Just give them without the : and follow them with a CR.

In rare instances, an internal error may occur in *vi*. In this case you will get a diagnostic and be left in the command mode of *ex*. You can then save your work and quit if you wish by giving a command **x** after the : which *ex* prompts you with, or you can re-enter *vi* by giving *ex* a *vi* command.

There are a number of things which you can do more easily in *ex* than in *vi*. Systematic changes in line-oriented material are particularly easy. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing. Keep these things in mind as you read on.

2.2. Getting Started

When using *vi*, changes which you make to the file you are editing are reflected in what you see on your workstation screen.

[1] The material in this chapter is derived from *An Introduction to Display Editing with Vi*, W.N. Joy, M. Horton, University of California, Berkeley and *Vi Command and Function Reference*, A.P.W. Hewett, M. Horton.

During an editing session, there are two usual modes of operation: *command* mode and *insert* mode. In command mode you can move the cursor around in the file. There are commands to move the cursor forward and backward in units of characters, words, sentences and paragraphs. A small set of operators, like **d** for delete and **c** for change, are combined with the motion commands to form operations such as delete word or change paragraph. You can do other operations which don't involve entering fresh text. To enter new text into the file, you must be in insert mode, which you get with the **a** (append), **o** (open) and **i** (insert) commands. You get out of insert mode by typing the ESC (escape) key (or ALT on some keyboards). The significant characteristic of insert mode is that commands can't be used, so anything you type except ESC is inserted into the file. If you change your mind anytime using *vi*, typing 2ESC cancels the command you started and reverses to command mode. Also, if you are unsure of which mode you are in, type ESC until the screen flashes; this means that you are back in command mode.

Run *vi* on a copy of a file you are familiar with while you are reading this. Try the commands as they are described.

2.2.1. Editing a File

To use *vi* on the file, type:

logo% **vi** *filename*

replacing *filename* with the name of the file copy you just created. The screen clears and the text of your file appears.

If you do not get the display of text, you may have typed the wrong filename. *Vi* has created a new file for you with the indication ' "file" [New file]'. Type **:q** (colon and the 'q' key) and then type the RETURN key. This should get you back to the command level interpreter. Then try again, this time spelling the filename correctly.

If *vi* doesn't seem to respond to the commands which you type here, try sending an interrupt to it by typing a ^C (or INTERRUPT signal) at your workstation (or by pressing the DEL or RUB keys on your terminal). Then type the **:q** command again followed by a RETURN. If you are using a terminal and something else happens, you have may given the system an incorrect terminal type code. *Vi* may make a mess out of your screen. This happens when it sends control codes for one kind of terminal to some other kind of terminal. Type a **:q** and RETURN. Figure out what you did wrong (ask someone else if necessary) and try again.

2.2.2. The Editor's Copy — Editing in the Buffer

Vi does not directly modify the file which you are editing. Rather, *vi* makes a copy of this file in a place called the *buffer*, and remembers the file's name. All changes you make while editing only change the contents of the buffer. You do not affect the contents of the file unless and until you *write* the buffer back into the original file.

2.2.3. Arrow Keys

The editor command set is independent of the workstation or terminal you are using. On most terminals with cursor positioning keys, these keys will also work within the editor.[2] If you don't have cursor positioning keys, that is, keys with arrows on them, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys. As you will see later, **h** moves back to the left

(like ^H which is a backspace), j moves down (in the same column), k moves up (in the same column), and l moves to the right.

2.2.4. Special Characters: ESC, CR and ^C

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labelled ESC (or ALT on some terminals). It is near the upper left corner of your workstation keyboard. Try typing this key a few times. *Vi* flashes the screen (or beeps) to indicate that it is in a quiescent state. You can cancel partially formed commands with ESC. When you insert text in the file, you end the text insertion with ESC. This key is a fairly harmless one to press, so you can just press it until the screen flashes if you don't know what is going on.

Use RETURN (or CR for *carriage return*) key to terminate certain commands. It is at the right side of the workstation keyboard, and is the same key used at the end of each Shell command.

Use the special character ^C (or DEL or RUB key), which sends an interrupt, to tell *vi* to stop what it is doing. It is a forceful way of making *vi* listen to you, or to return it to the quiescent state if you don't know or don't like what is going on.

Try typing the '/' key on your keyboard. Use this key to search for a string of characters. *Vi* displays the cursor at the bottom line of the screen after a '/' is displayed as a prompt. You can get the cursor back to the current position by pressing RETURN (or ESC or DEL); try this now. Backspacing over the '/' will also cancel the search. From now on we will simply refer to typing ^C (or pressing the DEL or RUB key) as 'sending an interrupt.'[3]

Vi often echoes your commands on the last line of the screen. If the cursor is on the first position of this last line, then *vi* is performing a computation, such as locating a new position in the file after a search or running a command to reformat part of the buffer. When this is happening, you can stop *vi* by sending an interrupt.

2.2.5. Getting Out of vi — :q, :q!, :w, ZZ, :wq

When you want to get out of *vi* and end the editing session, type :q to *quit*. If you have changed the buffer contents and type :q, *vi* responds with 'No write since last change (:quit! overrides).' If you then want to quit *vi* without saving the changes, type :q!. You need to know about :q! in case you change the editor's copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

Do not type :q! if you *want* to save your changes. To save or *write* your changes without quitting *vi*, type :w. If you are sure about some changes in the middle of an editing session, it's a good idea to save your changes from time to time.

To write the contents of the buffer back into the file you are editing, with any changes you have made, and then to quit, type ZZ. And finally, to write the file even if no changes have been made, and exit *vi*, type :wq.

---

[2] Note for the HP2621: on this terminal the function keys must be *shifted* (ick) to send to the machine, otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.

[3] On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.

You can terminate all commands which read from the last display line with an ESC as well as a RETURN.

2.3. Moving Around in the File

Vi has a number of commands for moving around in the file. You can *scroll* forward and backward through a file, moving part of the text on the screen. You can *page* forward and backward through a file, by moving a whole screenfull of text. You can also display one more line at the top or bottom of the screen.

2.3.1. Scrolling and Paging — ^D, ^U, ^E, ^Y, ^F, ^B

The most useful way to move through a file is to type the control (CTRL) and D keys at the same time, sending a control-D or '^D'. We use this two-character notation to refer to control keys from now on. The shift key is ignored, so ^D and ^d are equivalent. If you are using a terminal, you may have a key labelled '^' on your keyboard. This key is represented as '^' and is used exclusively as part of the '^x' notation for control characters.[4]

Try typing **^D** to see that this command scrolls *down* in the file. The command to scroll *up* is **^U**. (Many dumb terminals can't scroll up at all, in which case type **^U** to clear and refresh the screen with a line which is farther back in the file at the top.)

If you want to see more of the file below where you are, you can type **^E** to *expose* one more line at the bottom of the screen, leaving the cursor where it is. The **^Y** (which is hopelessly non-mnemonic, but next to **^U** on the keyboard) exposes one more line at the top of the screen.

You can also use the keys **^F** and **^B** to move *forward* and *backward* a page, keeping a couple of lines of continuity between screens so that it is possible to read through a file using these rather than **^D** and **^U** if you wish. **^F** and **^B** also take preceding counts, which specify the number of pages to move. For example, **2^F** pages forward two pages.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, typing **^F** to page forward leaves you only a little context to look back at. Scrolling with **^D** on the other hand, leaves more context, and moves more smoothly. You can continue to read the text as scrolling is taking place.

2.3.2. Searching, Goto, and Previous Context — /, ?, G

Another way to position yourself in the file is to give *vi* a string to search for. Type the character '**/**' followed by a string of characters terminated by RETUR. *Vi* positions the cursor at the next occurrence of this string. Try typing **n** to then go to the *next* occurrence of this string. The character '**?**' searchs backwards from where you are, and is otherwise like '**/**'. **N** is like **n**, but reverses the direction of the search.

You can string several search expressions together, separated by a semicolon in visual mode, the same as in command mode in *ex*. For example:

/today/;/tomorrow

[4] If you don't have a CTRL or '^' key on your terminal, there is probably a key labelled '↑'; in any case these characters are one and the same.

moves the cursor to the first 'tomorrow' after the next 'today'. This also works within one line.

These searches normally wrap around the end of the file, so you can find the string even if it is not on a line in the direction you search, but provided it is somewhere else in the file. You can disable this *wraparound* with the command **:se nowrapscan**CR, or more briefly **:se nows**CR.

If the search string you give *vi* is not present in the file, *vi* displays 'Pattern not found' on the last line of the screen, and the cursor is returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with a '^'. To match only at the end of a line, end the search string with a '$'. So to search for the word 'search' at the beginning of a line, type:

/^searchCR

and to search for the word 'last' at the end of a line, type:

/last$CR

Actually, the string you give to search for here can be a *regular expression* in the sense of the editors *ex* and *ed*. If you don't wish to learn about this yet, you can disable this more general facility by typing **:se nomagic**CR; by putting this command in EXINIT in your environment, you can have always this *nomagic* option in effect. See *Special Topics* for details on how to do this.

The command **G**, when preceded by a number positions the cursor at that line in the file. Thus **1G** moves the cursor to the first line of the file. If you do not give **G** any count, it positions you at the last line of the file.

If you are near the end of the file, and the last line is not at the bottom of the screen, *vi* places only the character '~' on each remaining line. This indicates that the last line in the file is on the screen; that is, the '~' lines are past the end of the file.

You can find out the state of the file you are editing by typing a **^G**. *Vi* shows you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and the percentage of characters already displayed from the buffer. For example:

"data.file" [Modified] line 329 of 1276 --8%--

Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end and then another **G** command with the line number to get back where you were.

You can get back to a previous position by using the command '' (two apostrophes). This returns you to the first non-white space in the previous location. You can also use `` (two back quotes) to return to the previous position. The former is more easily typed on the keyboard. This is often more convenient than **G** because it requires no advance preparation. Try typing a **G** or a search with **/** or **?** and then a '' to get back to where you were. If you accidentally type **n** or any command which moves you far away from a context of interest, you can quickly get back by typing ''.

2.3.3. Moving Around on the Screen — h, j, k, l

Now try just moving the cursor around on the screen. Try the arrow keys as well as **h**, **j**, **k**, and **l**. You will probably prefer these keys to arrow keys, because they are right underneath your fingers. These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys then the screen scrolls down (and up if possible) to bring a line at a time into view.

Type the + key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The – key is like + but goes the other way.

The RETURN key has the same effect as the + key.

Vi also has commands to take you to the top, middle and bottom of the screen. **H** takes you to the top (*home*) line on the screen. Try preceding it with a number as in **3H**. This takes you to the third line on the screen. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the *last* line on the screen. **L** also takes counts, so **5L** takes you to the fifth line from the bottom.

2.3.4. Moving Within a Line — b, w, e, B, W

Now pick a word on some line on the screen, not the first word on the line. Move the cursor using **h**, **j**, **k**, **l** or RETURN and – to be on the line where the word is. Try typing the **w** key. This advances the cursor to the next *word* on the line. Try typing the **b** key to *back* up words in the line. Also try the **e** key which advances you to the *end* of the current word rather than to the beginning of the next word. Also try SPACE (the space bar) which moves right one character and the BACKSPACE (or **^H**) key which moves left one character. The key **h** works as **^H** does and is useful if you don't have a BACKSPACE key. Also, as noted just above, **l** moves to the right.

If the line had punctuation in it, you may have noticed that the **w** and **b** keys stopped at each group of punctuation. You can also go back and forwards words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly typing **w**.

2.3.5. Viewing a File — 'view'

If you want to use the editor to look at a file, rather than to make changes, use *view* instead of *vi*. This sets the *readonly* option which prevents you from accidently overwriting the file. For example, to look at a file called *kubla*, type:

```
logo% view kubla
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.
"kubla" [Read only] 5 lines, 149 characters
logo%
```

To scroll through a file that is bigger than one screen, use the characters described in *Scrolling and Paging*. To get out of *view*, type **:q**.

2.4. Making Simple Changes

Simple changes involve inserting, deleting, repeating, and changing single characters, words, and lines of text. In *vi*, you can also undo the previous change with ease in case you change your mind.

2.4.1. Inserting — i and a

There are two basic commands for inserting new text: **i** *inserts* text to the left of the cursor, and **a** to *appends* text to the right of the cursor. After you type **i**, everything you type until you press ESC is inserted into the file. Try this now; position yourself at some word in the file and try inserting text before this word. (If you are on an dumb terminal it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you type ESC.)

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and ESC to terminate the textual insert. Use this sequence of commands to easily pluralize a word.

Try inserting and appending a few times to make sure you understand how this works.

It is often the case that you want to add new lines to the file you are editing, before or after some specific line in the file. Find a line where this makes sense and then give the command **o** to create a new line *after* the line you are on, or the command **O** to create a new line *before* the line you are on. After you create a new line in this way, text you type up to an ESC is inserted on the new line.

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower-case key and the other is given by an upper-case key. In these cases, the upper-case key often differs from the lower-case key in its sense of direction, with the upper-case key working backward and/or up, while the lower-case key moves forward and/or down.

Whenever you are typing in text, you can give many lines of input or just a few characters. To type in more than one line of text, type a RETURN at the middle of your input. A new line will be created for text, and you can continue to type. (If you are on a slow, dumb terminal *vi* may choose to wait to redraw the tail of the screen, and will let you type over the existing screen lines. This avoids the lengthy delay which would occur if *vi* attempted to always keep the tail of the screen up to date. The tail of the screen will be fixed up, and the missing lines will reappear, when you type ESC.)

While you are inserting new text, you can use the DEL key at the system command level to backspace over the last character which you typed. (This may be **^H** on a terminal.) Use **^U** (this may be **^X** on a terminal) to erase the input you have typed on the current line. In fact, the character **^H** (backspace) always works to erase the last input character here, regardless of what your erase character is.

^W erases a whole word and leaves you after the space after the previous word; use it to quickly back up when inserting.

Notice that when you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you press ESC; if you want to get rid of them immediately, hit an ESC and then **a** again.

Notice also that you can't erase characters which you didn't insert, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a

correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

2.4.2. Making Small Corrections — x, r, x, R

You can make small corrections in existing text quite easily. Find a single character which is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace with **h** (or the BACKSPACE key or ^H) or type a SPACE (using the space bar) until the cursor is on the character which is wrong. If the character is not needed, type the **x** key; this deletes the character from the file. It is analogous to the way you x out characters when you make mistakes on a typewriter, except it's not as messy.

If the character is incorrect, you can *replace* it with the correct character by typing the command **r***c*, where *c* is replaced by the correct character. You don't need to type ESC. Finally if the character which is incorrect should be replaced by more than one character, type **s** which *substitutes* a string of characters, ending with ESC, for it. If there are a small number of characters which are wrong you can precede **s** with a count of the number of characters to be replaced. You can use counts with **x** to specify the number of characters to be deleted and with **r**, such as **4rx** to specify that a character be replaced with four x's.

Use **xp** to correct simple typos in which you have inverted the order of two letters. The **p** for *put* is described later.

2.4.3. Deleting, Repeating, and Changing — dw, ., db, c

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to *delete* a *word*. Try typing '.' a few times. Notice that this repeats the effect of the **dw**. The '.' repeats the last command which made a change. You can remember it by analogy with an ellipsis '...'.

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

Use **D** to delete the rest of the line the cursor is on

Another very useful operator is **c** or *change*. Thus **cw** changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word which you can change to another, and try this now. Notice that the end of the text to be changed is marked with the character '$' so that you can see this as you are typing in the new material.

2.4.4. Operating on Lines — dd, cc, S

It is often the case that you want to operate on lines. Find a line which you want to delete, and type **dd**, the **d** operator twice. This deletes the line.

If you are on a dumb terminal, *vi* may just erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this changes a whole line, erasing its previous contents and replacing them with text you type up to an ESC. The command **S** is a convenient synonym for

cc, by analogy with **s**. Think of **S** as a substitute on lines, while **s** is a substitute on characters.

You can delete or change more than one line by preceding the **dd** or **cc** with a count, such as **5dd**, which deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the *last* line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.[5] Notice that *vi* lets you know when you change a large number of lines so that you can see the extent of the change. It also always tells you when a change you make affects text which you cannot see.

2.4.5. Undoing — u, U

Now suppose that the last change which you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since it is often the case that we regret a change or make a change incorrectly, *vi* provides a **u** command to *undo* the last change which you made. Try this a few times, and give it twice in a row to notice that an **u** also undoes a **u**.

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text which you delete, even if undo will not bring it back; see *Recovering Lost Lines* on how to recover lost text.

2.5. Moving About: Rearranging and Duplicating Text

This describes more commands for moving in a file and explains how to rearrange and make copies of text.

2.5.1. Low-level Character Motions — f, F, ^

Now move the cursor to a line where there is a punctuation or a bracketing character such as a parenthesis, a comma or a period. Try the command **f***x* to *find* the next *x* character to the right of the cursor in the current line. Try then hitting a **;** which finds the next instance of the same character. By using the **f** command and then a sequence of **;**'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. There is also an **F** command, which is like **f**, but searches backward. The **;** also repeats **F**.

When you are operating on the text in a line, it is often desirable to deal with the characters up to, but not including, the first instance of a character. Try **df***x* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x*; the **t** here stands for *to*, that is, delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

When working with the text of a single line, a '^' moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus **$a** appends new text at the end of the current line (as does **A** which is easier to type).

---

[5] One subtle point here involves using the '/' search after a **d**. This normally deletes characters from the current position to the point of the match. If what is desired is to delete whole lines including the two points, give the pattern as */pat/+0*, a line address.

Your file may have tab (^I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every eight positions.[6] When the cursor is at a tab, it sits on the last of the several spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have non-printing characters in it. These characters are displayed in the same way they are represented in this chapter, that is with a two-character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but spacing or backspacing over the character reveals that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes discards control characters, depending on the character and the setting of the *beautify* option, if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a ^V before the control character. The ^V quotes the following character, causing it to be inserted directly into the file.

2.5.2. Higher Level Text Objects — (,), {, }, [[,]]

In working with a document it is often advantageous to work in terms of sentences, paragraphs, and sections. The operations '(' and ')' move to the beginning of the previous and next sentences respectively. Thus the command **d)** deletes the rest of the current sentence; likewise **d(** deletes the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

A sentence is defined as ending at a '.', '!' or '?' which is followed by either the end of a line, or by two spaces. Any number of closing ')', ']', '"' and ''' characters may appear after the '.', '!' or '?' before the spaces or end of line.

The operations '{' and '}' move over paragraphs and the operations '[[' and ']]' move over sections. The '[[' and ']]' operations require the operation character to be doubled because they can move the cursor far from where it currently is. While it is easy to get back with the command '``', these commands would still be frustrating if they were easy to type accidentally.

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option *paragraphs*. The default setting for this option defines the paragraph macros of the *-ms* macro package, that is the '.IP', '.LP', '.PP' and '.QP' macros. You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See *Special Topics* for details. The '.bp' directive is also considered to start a paragraph. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands take counts to operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally '.NH' and '.SH', and each line with a formfeed ^L in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different view size in which to redraw the screen at the new location, and this size is the base size for newly drawn screens until another size is specified. (This is very useful if you are on a slow terminal and are looking for a

[6] You can set this with a command of the form :se ts=*x*CR, where *x* is four to set tabstops every four columns, for example. This affects the screen representation within the editor.

particular section. You can give the first section command a small count to then see each successive section heading in a small screen area.)

2.5.3. Rearranging and Duplicating Text — y, p, P

Vi has a single unnamed buffer where the last deleted or changed away text is saved, and a set of named buffers **a–z** which you can use to save copies of text and to move text around in your file and between files.

The operator **y** *yanks* a copy of the object which follows into the unnamed buffer. If preceded by a buffer name, "*x***y**, where *x* here is replaced by a letter **a–z**, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P**; **p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text which you yank forms a part of a line, or is an object such as a sentence which partially spans more than one line, then when you put the text back, it will be placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like an **o** or **O** command.

Try the command **YP**. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** will also make a copy of the current line, and place it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place, and put it back in another. You can precede a delete operation by the name of a buffer in which the text is to be stored as in "**a5dd** deleting 5 lines into the named buffer **a**. You can then move the cursor to the eventual resting place of the lines and do a "**ap** or "**aP** to put them back. In fact, you can switch and edit another file before you put the lines back, by giving a command of the form **:e** *name*CR where *name* is the name of the other file you want to edit. You will have to write back the contents of the current editor buffer (or discard them) if you have made changes before *vi* will let you switch to the other file. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files, so to move text from one file to another you must use a named buffer.

2.6. High-Level Commands

A description of high-level commands that do more than juggle text follows.

2.6.1. Writing, Quitting, Editing New Files — ZZ, :w, :q, :e, :n

So far you have seen how to enter *vi* and to write out your file using either **ZZ** or **:w**CR. The first exits from *vi*, writing if changes were made, and the second writes and stays in *vi*. We have also described that if you have changed the editor's copy of the file but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement to the file, you type **:q!**CR to *quit* from the editor without writing the changes.

You can also re-*edit* the same file and start over by typing **:e!**CR. Use the '!' command rarely and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can also edit a different file without leaving *vi* by giving the command **:e** *name*CR. If you have not written out your file before you try to do this, *vi* tells you this, ('No write since last change: (:edit! overrides)') and delays editing the other file. You can then type **:w**CR to save your work, followed by the **:e** *name*CR command again, or carefully give the command **:e!** *name*CR, which edits the other file discarding the changes you have made to the current file. To save changes automatically, include *set autowrite* in your EXINIT, and use **:n** instead of **:e**. See *Special Topics* for details on EXINIT.

2.6.2. Escaping to a Shell — :!, :sh, ^Z

You can get to a Shell to execute a single command by giving a *vi* command of the form **:!***cmd*CR. The system runs the single command *cmd* and when the command finishes, *vi* asks you to 'Press RETURN to continue.' When you have finished looking at the output on the screen, type RETURN, and *vi* redraws the screen. You can then continue editing. You can also give another **:** command when it asks you for a RETURN; in this case the screen will not be redrawn.

If you wish to execute more than one command in the Shell, give the command **:sh**CR. This gives you a new Shell, and when you finish with the Shell, ending it by typing a **^D**, *vi* clears the screen and continues.

Use **^Z** to suspend *vi* and to return to the top level Shell. The screen is redrawn when *vi* is resumed. This is the same as **:stop**.

2.6.3. Marking and Returning — m

The command `` returned to the previous place after a motion of the cursor by a command such as **/**, **?** or **G**. You can also *mark* lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and type **`a**. The cursor will return to the place which you marked. Marks last only until you edit another file.

When using operators such as **d** and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by **m**. In this case you can use the form **'***x* rather than **`***x*. Used without an operator, **'***x* will move to the first non-white character of the marked line; similarly **''** moves to the first non-white character of the line containing the previous context mark ``.

2.6.4. Adjusting the Screen ^L, z

If the screen image is messed up because of a transmission error to your workstation, or because some program other than *vi* wrote output to your workstation, you can type a **^L**, the ASCII form-feed character, to refresh the screen. (On a dumb terminal, if there are **@** lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing **^R** to *retype* the screen, closing up these holes.[7])

If you wish to place a certain line on the screen at the top middle or bottom of the screen, position the cursor to that line, and give a **z** command. Follow the **z** command with a RETURN if

[7] This includes Televideo 912/920 and ADM31 terminals.

you want the line to appear at the top of the window, a '.' if you want it at the center, or a '–' if you want it at the bottom.

If you want to change the window size, use the **z** command as in **z5**<CR> to change the window to five lines.

2.7. Special Topics

There are several facilities that you can use to customize an editing session.

2.7.1. Options, the Set Variable, and Editor Start-up Files

vi has a set of options, some of which have been mentioned above. The most useful options are described in the following table.

Table 2-1: Editor Options

Option	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before :n, :ta, ^↑, !
ignorecase	noic	Ignore letter case in searching
lisp	nolisp	({) } commands deal with S-expressions
list	nolist	Tabs print as ^I; end of lines marked with $
magic	nomagic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names which start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one
sections	sect=NHSHH HU	Macro names which start new sections
shiftwidth	sw=8	Shift distance for <, > and input ^D and ^T
showmatch	nosm	Show matching (or { as) or } is displayed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form:

set *opt*=*val*

and toggle options can be set or unset by statements of one of the forms

set *opt*
set no*opt*

Put these statements in your EXINIT in your environment (described below), or use them while you are running *vi* by preceding them with a : and following them with a RETURN. For example, to display line numbers, use:

:se nu

You can get a list of all options which you have changed:

```
:setCR
redraw term=sun wrapmargin=8
```

or the value of a single option with **:set** *opt*?CR:

```
:set noai?cr
noautoindent
```

The **:set all**CR command generates a list of all possible options and their values. You can abbreviate set to **se**. You can also put multiple options on one line, such as, **:se ai aw nu**CR.

When you set options with the **set** command, they only last while you stay in *vi*. It is common to want to have certain options set whenever you use the editor. To do this, create a list of *ex* commands to be run every time you start up *vi*, *ex*, or *edit*. All commands which start with **:** are *ex* commands. A typical list includes a **set** command, and possibly a few **map** commands. Put these commands on one line by separating them with the '|' character. If you use *csh*, put a line like this in the *.login* file in your home directory:

```
setenv EXINIT 'set ai aw terse|map @ dd|map # x'
```

which sets the options *autoindent*, *autowrite*, *terse*, (the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). (See the *Macros* section for a description of the **map** command.)

If you use the Bourne Shell, put these lines in the file *.profile* in your home directory:

```
EXINIT='set ai aw terse|map @ dd|map # x'
export EXINIT
```

Of course, the particulars of the line would depend on which options you wanted to set.

2.7.2. Recovering Lost Lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, *vi* saves the last nine deleted blocks of text in a set of numbered registers 1–9. You can get the *n*'th previous deleted text back in your file by "*n*p. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and p, which *puts* text in the buffer after the cursor. If this doesn't bring back the text you wanted, type u to undo this and then . (period) to repeat the p. In general the '.' command repeats the last change you made. As a special case, when the last command refers to a numbered text buffer, the '.' command increments the number of the buffer before repeating the command. Thus a sequence of the form:

```
"1pu.u.u.
```

will, if repeated long enough, show you all the deleted text which has been saved for you. You can omit the u commands here to gather up all this text in the buffer, or stop after any . command to keep just the recovered text. You can also use P rather than p to put the recovered text before rather than after the cursor.

2.7.3. Recovering Lost Files — the –r Option

If something goes wrong so the system goes down, you can recover the work you were doing up to the last few changes. You will normally receive mail when you next log in giving you the name of the file which has been saved for you. You should then change to the directory where

you were when the system went down and type:

```
logo% vi -r name
```

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off. In rare cases, some of the lines of the file may be lost. *Vi* will give you the numbers of these lines and the text of the lines will be replaced by the string 'LOST'. These lines will almost always be among the last few which you changed. You can either choose to discard the changes which you made (if they are easy to remake) or to replace the few lost lines by hand.

You can get a listing of the files which are saved for you by typing:

```
logo% vi -r
```

If there is more than one instance of a particular file saved, *vi* gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

The invocation '*vi -r*' will not always list all saved files, but they can be recovered even if they are not listed.

2.7.4. Continuous Text Input — wrapmargin

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. To do this, use the *set wrapmargin* option:

```
:se wm=10CR
```

This rewrites words on the next line that you type past the right margin.

If *vi* breaks an input line and you wish to put it back together, you can tell it to join the lines with **J**. You can give **J** a count of the number of lines to be joined as in **3J** to join 3 lines. *Vi* supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can delete the white space with **x** if you don't want it.

If you want to *split* a line into two, put the cursor where you want the break, and type **r**CR.

2.7.5. Features for Editing Programs

Vi has a number of commands for editing programs. To enable the *autoindent* facility for helping you generate correctly indented programs, use the *autoindent* option:

```
:se aiCR.
```

Now try opening a new line with **o** and type some characters on the line after a few tabs. If you now start another line, notice that *vi* supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **^D** key to backtab over the supplied indentation.

Each time you type **^D**, you back up one position, normally to an eight-column boundary. You can set this amount with the *shiftwidth* option, which changes this value. Try giving the command:

```
:se sw=4CR
```

and then experimenting with autoindent again.

For shifting lines in the program left and right, there are operators < and >. These shift the lines you specify right or left by one *shiftwidth*. Try << and >> which shift one line left or right, and <L and >L shifting the rest of the text left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and type %. This shows you the matching parenthesis. This works also for braces { and }, and brackets [and].

If you are editing C programs, you can use [[and]] to advance or retreat to a line starting with a {, that is, a function declaration at a time. When you use]] with an operator, it stops after a line which starts with }; this is sometimes useful with y]].

2.7.6. Filtering Portions of the Buffer

You can run system commands over portions of the buffer using the operator '!'. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command:

!}sortCR.

This says to sort the next paragraph of material, and that the blank line ends a paragraph. The result is sorted text in your file.

2.7.7. Commands for Editing LISP

If you are editing a LISP program, set the option *lisp* by doing:

:se lispCR.

This changes the (and) commands to move backward and forward over s-expressions. The { and } commands are like (and) but don't stop at atoms. Use { and } to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying indentation to align at the first argument to the last open list. If there is no such argument, the indent is two spaces more than the last level.

The *showmatch* option shows matching parentheses. Try setting it with:

:se smCR

and then try typing a '(' *some words* and then a ')'. Notice that the cursor briefly shows the position of the '(' which matches the ')'. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

Vi also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the == operator. Try the command ==% at the beginning of a function. This realigns all the lines of the function declaration.

When you are editing LISP, the [[and]] advance and retreat to lines beginning with a (, and are useful for dealing with entire function definitions.

2.7.8. Macros

Vi has a parameterless macro facility, which you can set up so that when you type a single keystroke, *vi* will act as though you had typed some longer sequence of keys. Set this up if you find yourself repeatedly typing the same sequence of commands or text.

Briefly, there are two kinds of macros:

1. Ones where you put the macro body in a buffer register, say *x*. You can then type @x to invoke the macro. The @ may be followed by another @ to repeat the last macro.
2. You can use the *map* command from *vi* (typically in your EXINIT) with a command of the form:

   :map *lhs rhs*CR

   mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either one character or one function key) since it must be entered within one second unless *notimeout* (see *Option Descriptions*) is set, in which case you can type it as slowly as you wish, and *vi* will wait for you to finish it before it echoes anything). The *lhs* can be no longer than ten characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs*, escape them with a ^V. It may be necessary to double the ^V if you use the map command inside *vi*, rather than in *ex*. You do not need to escape spaces and tabs inside the *rhs*.

Thus to make the q key write and exit *vi*, type:

:map q :wq^V^VCR CR

which means that whenever you type **q**, it will be as though you had typed the four characters **:wq**CR. A ^V is needed because without it the CR would end the **:** command, rather than becoming part of the *map* definition. There are two ^V's because from within *vi*, you must type two ^V's to get one. The first CR is part of the *rhs*, the second terminates the **:** command.

You can delete macros with

:unmap *lhs*

If the *lhs* of a macro is '#0' through '#9', this maps the particular function key instead of the two-character '#' sequence. So that terminals without function keys can access such definitions, the form '#x' will mean function key *x* on all terminals and need not be typed within one second. You can change the character '#' by using a macro in the usual way:

:map ^V^V^I #

to use tab, for example. This won't affect the *map* command, which still uses #, but just the invocation from visual mode.

The *undo* command reverses an entire macro call as a unit, if it made any changes.

Placing a 'l' after the word **map** applies the mapping to input mode, rather than command mode. So, to arrange for ^T to be the same as four spaces in input mode, type:

:map ^T ^V␣␣␣␣

where ␣ is a blank. The ^V prevents the blanks from being taken as white space between the *lhs* and *rhs*. Type simply:

:map!

to list macros that apply during input mode and

:map

to list macros that apply during command mode. Typing **map** or **map!** by itself produces a listing of macros in the corresponding mode.

2.7.9. Word Abbreviations — :ab, :una

A feature similar to macros in input mode is word abbreviation. You can type a short word and have it expanded into a longer word or words with **:abbreviate (:ab)**. For example:

:ab foo find outer otter

always changes the word 'foo' into the phrase 'find outer otter'. Word abbreviation is different from macros in that only whole words are affected. If 'foo' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. There is no need for an abbreviation to be a single keystroke, as it should be with a macro. This only operates in visual mode and uses the same syntax as the *map* command, except that there are no '!' forms.

Use **:unabbreviate (:una)** to turn off the abbreviation. To unabbreviate the above, for example, type:

:una foo

2.7.9.1. Abbreviations

Vi editor has a number of short commands which abbreviate the longer commands have introduced here. You can find these commands easily in the *Ex Commands* section in the quick reference. They often save a bit of typing, and you can learn them when it's convenient.

2.8. Nitty-gritty Details

The following presents some functional details and some *ex* commands (see *File Manipulation Commands*) that are particularly useful in *vi*.

2.8.1. Line Representation in the Display

Vi folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and skip over all the segments of a line in one motion. The command **|** moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line which is more than 80 columns long. You can make long lines very easily by using **J** to join together short lines.

Vi only puts full lines on the display; if there is not enough room on the display to fit a logical line, *vi* editor leaves the physical line empty, placing only an '@' on the line as a place holder. (When you delete lines on a dumb terminal, *vi* will often just clear the lines to '@' to save time rather than rewriting the rest of the screen.) You can always maximize the information on the screen with **^R**.

If you wish, you can have the editor place line numbers before each line on the display. To enable this, type the option:

:se nuCR

To turn it off, use the *no numbers* option:

:se nonuCR

You can have tabs represented as ^I and the ends of lines indicated with '$' by giving the *list* option:

:se listCR

To turn this off, use:

:se nolistCR

Finally, lines consisting of only the character '~' are displayed when the last line in the file is in the middle of the screen. These represent physical lines which are past the logical end of file.

2.8.2. Command Counts

Most *vi* commands use a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

New window size	**: / ? [[ ]] ` '**
Scroll amount	**^D ^U**
Line/column number	**z G |**
Repeat effect	Most of the rest

Vi maintains a notion of the current default window size. (On terminals which run at speeds greater than 1200 baud *vi* uses the full terminal screen. On terminals which are slower than 1200 baud, and most dialup lines are in this group, *vi* uses 8 lines as the default window size. At 1200 baud the default is 16 lines.)

Vi uses the default window size when it clears and refills the screen after a search or other motion moves far from the edge of the current window. The commands which take a new window size as count all often redraw the screen. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a '–' or similar command or off the bottom with a command such as RETURN or **^D**. The window will revert to the last specified size the next time it is cleared and refilled, but not by a **^L** which just redraws the screen as it is.

The scroll commands **^D** and **^U** likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a+**———ESC will insert a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for the few commands which ignore any counts, such as **^R**, the rest of the *vi* commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**RETURN advances five lines. A very useful instance of a count as a repetition is a count given to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you delete first one and then three words. You can then delete two more words with **2.**.

2.8.3. File Manipulation Commands

The following table lists the file manipulation commands which you can use when you are in *vi*.

Table 2-2: File Manipulation Commands

Command	Meaning
:w	Write back changes
:wq	Write and quit
:x	Write (if necessary) and quit (same as ZZ).
:e *name*	Edit file *name*
:e!	Re-edit, discarding changes
:e + *name*	Edit, starting at end
:e +*n*	Edit, starting at line *n*
:e #	Edit alternate file
:w *name*	Write file *name*
:w! *name*	Overwrite file *name*
:*x*,*y*w *name*	Write lines *x* through *y* to *name*
:r *name*	Read file *name* into buffer
:r !*cmd*	Read output of *cmd* into buffer
:n	Edit next file in argument list
:n!	Edit next file, discarding changes to current
:n *args*	Specify new argument list
:ta *tag*	Edit file containing tag *tag*, at *tag*

A CR or ESC follows all of these commands. The most basic commands are :w and :e. End a normal editing session on a single file with a ZZ command. If you are editing for a long period of time, use the :w command occasionally after major amounts of editing, and then finish with a ZZ. When you edit more than one file, you can finish with one with a :w and start editing a new file by giving a :e command, or set *autowrite* and use :n *file*.

If you make changes to the editor's copy of a file, but do not wish to write them back, give an ! after the command you would otherwise use to exit without changing the file. Use this carefully.

Use the :e command with a + argument to start at the end of the file, or a +*n* argument to start at line *n*. In actuality, *n* may be any editor command not containing a space, usually a scan like +/*pat* or +?*pat*. In forming new names to the e command, use the character % which is replaced by the current filename, or the character # which is replaced by the alternate filename. The alternate filename is generally the last name you typed other than the current file. Thus if you try to do a :e and get a diagnostic that you haven't written the file, you can give a :w command and then a :e # command to redo the previous :e.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using ^G, and giving these numbers after the : and before the w, separated by ,'s. You can also mark these lines with m and then use an address of the form 'x,'y on the w command here.

You can read another file into the buffer after the current line by using the :r command. You can similarly read in the output from a command, just use !*cmd* instead of a filename.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command :n. To respecify the list of files to be edited,

give the **:n** command a list of filenames, or a pattern to be expanded as you would have given it on the initial *vi* command.

For editing large programs, use the **:ta** command. It utilizes a data base of function names and their locations, which can be created by programs such as *ctags*, (see the *User's Manual for the Sun Workstation*) to quickly find a function whose name you give. If the **:ta** command will require the editor to switch files, then you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again.

2.8.4. More about Searching for Strings

When you are searching for strings in the file with **/** and **?**, *vi* normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c** or **y**, then you may well wish to affect lines up to the line before the line containing the pattern. You can give a search of the form */pat/–n* to refer to the *n*'th line before the next line containing *pat*, or you can use **+** instead of **–** to refer to the lines after the one containing *pat*. If you don't give a line offset, *vi* will affect characters up to the match place, rather than whole lines; thus use '+ 0' to affect to the line which matches.

To have *vi* ignore the case of words in searches, give the *ignorecase* option:

:se icCR

To turn this off so that *vi* recognizes case again, use:

:se noicCR

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should put:

set nomagic

in your EXINIT. In this case, only the characters '^' and '$' are special in patterns. The character \ is also then special, as it is most everywhere in the system, and you can use it to get at the extended pattern matching facility. It is also necessary to use a ' \' before a '/' in a forward scan or a '?' in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

Table 2-3: Extended Pattern Matching Characters

Character	Meaning
^	At beginning of pattern, matches beginning of line
$	At end of pattern, matches end of line
.	Matches any character
\<	Matches the beginning of a word
\>	Matches the end of a word
[*string*]	Matches any single character in *string*
[^ *string*]	Matches any single character not in *string*
[*x*–*y*]	Matches any character between *x* and *y*
*	Matches any number of the preceding pattern

If you use **nomagic** mode, use the '. [' and '*' primitives with a preceding \.

2.8.5. More about Input Mode

There are a number of characters to make corrections during input mode. These are summarized in the following table.

Table 2-4: Input Mode Corrections

Character	Meaning
^H	Deletes the last input character
^W	Deletes the last input word, defined as by b
erase	Your erase character, same as ^H
kill	Your kill character, deletes the input on this line
\	Escapes a following ^H and your erase and kill
ESC	Ends an insertion
DEL	Interrupts an insertion, terminating it abnormally
CR	Starts a new line
^D	Backtabs over *autoindent*
0^D	Kills all the *autoindent*
↑^D	Same as 0^D, but restores indent next line
^V	Quotes the next non-printing character into the file

The most usual way of making corrections to input is to type DEL (^H on a terminal) to correct a single character, or by typing one or more ^W's to back over incorrect words.

Your system kill character ^U (or sometimes ^X) erases all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor can you erase characters which you did not insert with this insertion command. To make corrections on the previous line after a new line has been started, press ESC to end the insertion, move over and make the correction, and then return to where you were to continue. Use **A** to append at the end of the current line; this is often useful for continuing text input.

If you wish to type in your erase or kill character, say ^U, you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a ^V. The ^V echoes as a ↑ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.[8]

If you are using *autoindent* you can backtab over the indent which it supplies by typing a ^D. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied *autoindent*.

When you are using *autoindent* you may wish to place a label at the left margin of a line. The way to do this easily is to type ^ and then ^D. The editor will move the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a **0** followed immediately by a ^D if you wish to kill all the indent and not have it come back on the next line.

[8] This is not quite true. *Vi* does not allow the NULL (^@) character to appear in files. Also the editor uses the LF (linefeed or ^J) character to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ^ before you type the character. In fact, the editor treats a following letter as a request for the corresponding control character. This is the only way to type ^S or ^Q, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

2.9. Command and Function Reference

The following section provides abridged explanations of the *vi* and *ex* commands.

2.9.1. Notation

Notation used in this section is as follows.

[option] Denotes optional parts of a command. Many vi commands have an optional count.

[cnt] Means that an optional number may precede the command to multiply or iterate the command.

{variable item}
Denotes parts of the command which must appear, but can take a number of different values.

<character [-character]>
Means that the character or one of the characters in the range described between the two angle brackets is to be typed. For example **<esc>** means type the **ESCAPE** key. **<a-z>** means type a lower-case letter. ^*<character>* means type the character as a **control** character, that is, with the CTRL key held down while simultaneously typing the specified character. Here we denote control characters with the *upper-case* character, but ^<uppercase chr> and ^<lowercase chr> are equivalent. That is, **^D** is equal to **^d**. The most common character abbreviations used in this list are as follows:

Table 2-5: Common Character Abbreviations

Character Abbreviation	Meaning
<esc>	escape, octal 033
<cr>	carriage return, ^M, octal 015
<lf>	linefeed ^J, octal 012
<nl>	newline, ^J, octal 012 (same as linefeed)
<bs>	backspace, ^H, octal 010
<tab>	tab, ^I, octal 011
<bell>	bell, ^G, octal 07
<ff>	formfeed, ^L, octal 014
<sp>	space, octal 040
<del>	delete, octal 0177

2.9.2. Commands

Following are brief explanations of the *vi* commands categorized by function for easy reference.

2.9.3. Entry and Exit

To enter *vi* on a particular *file*, type:

logo%vi *file*

The file will be read into the buffer, and the cursor is placed at the beginning of the first line. The first screenfull of the file is displayed on the screen.

To get out of the editor, type:

ZZ (or :q or :q!)

If you are in some special mode, such as input mode or the middle of a multi-keystroke command, it may be necessary to type ESC first.

2.9.4. Cursor and Page Motion

Note: You can move the cursor on your screen with the arrow keys on your workstation keyboard, the control character versions or the h, j, k, and l keys. If you are using a terminal that does not have arrow keys, use the control character versions or the h, j, k, and l keys.

[cnt]<bs> or [cnt]h or [cnt]←
: Move the cursor to the left one character. Cursor stops at the left margin of the page. [cnt] specifies the number of spaces to move.

[cnt]^N or [cnt]j or [cnt]↓ or [cnt]<lf>
: Move down one line. Moving off the screen scrolls the window to force a new line onto the screen. Mnemonic: Next

[cnt]^P or [cnt]k or [cnt]^
: Move up one line. Moving off the top of the screen forces new text onto the screen. Mnemonic: Previous

[cnt]<sp> or [cnt]l or [cnt]→
: Move to the right one character. Cursor will not go beyond the end of the line.

[cnt]-
: Move the cursor up the screen to the beginning of the next line. Scroll if necessary.

[cnt]+ or [cnt]<cr>
: Move the cursor down the screen to the beginning of the next line. Scroll up if necessary.

[cnt]$
: Move the cursor to the end of the line. If there is a count, move to the end of the line *cnt* lines forward in the file.

^
: Move the cursor to the beginning of the first word on the line.

0
: Move the cursor to the left margin of the current line.

[cnt]|
: Move the cursor to the column specified by the count. The default is column zero.

[cnt]w
: Move the cursor to the beginning of the next word. If there is a count, then move forward that many words and position the cursor at the beginning of the word. Mnemonic: next-word

[cnt]W Move the cursor to the beginning of the next word which follows a 'white space' (<sp>,<tab>, or <nl>). Ignore other punctuation.

[cnt]b Move the cursor to the preceding word. Mnemonic: backup-word

[cnt]B Move the cursor to the preceding word that is separated from the current word by a 'white space' (<sp>,<tab>, or <nl>).

[cnt]e Move the cursor to the end of the current word or the end of the *cnt*th word hence. Mnemonic: end-of-word

[cnt]E Move the cursor to the end of the current word which is delimited by 'white space' (<sp>,<tab>, or <nl>).

[line number]G Move the cursor to the line specified. Of particular use are the sequences 1G and G, which move the cursor to the beginning and the end of the file respectively. Mnemonic: Go-to

Note: The next four commands (^D, ^U, ^F, ^B) are not true motion commands, in that they cannot be used as the object of commands such as delete or change.

[cnt]^D Move the cursor down in the file by *cnt* lines (or the last *cnt* if a new count isn't given). The initial default is half a page. The screen is simultaneously scrolled up. Mnemonic: Down

[cnt]^U Move the cursor up in the file by *cnt* lines. The screen is simultaneously scrolled down. Mnemonic: Up

[cnt]^F Move the cursor to the next page. A count moves that many pages. Two lines of the previous page are kept on the screen for continuity if possible. Mnemonic: Forward-a-page

[cnt]^B Move the cursor to the previous page. Two lines of the current page are kept if possible. Mnemonic: Backup-a-page

[cnt](Move the cursor to the beginning of the next sentence. A sentence is defined as ending with a '.', '!', or '?' followed by two spaces or a <nl>.

[cnt]) Move the cursor backwards to the beginning of a sentence.

[cnt]} Move the cursor to the beginning of the next paragraph. This command works best inside *nroff* documents. It understands the *nroff* macros in *-ms*, for which the commands '.IP', '.LP', '.PP', '.QP', as well as the *nroff* command '.bp' are considered to be paragraph delimiters. A blank line also delimits a paragraph. The *nroff* macros that it accepts as paragraph delimiters are adjustable. See *Paragraphs* under *Set Commands*.

[cnt]{ Move the cursor backwards to the beginning of a paragraph.

]] Move the cursor to the next 'section,' where a section is defined by the set of *nroff* macros in *-ms*, in which '.NH' and '.SH' delimit a section. A line beginning with a <ff><nl> sequence, or a line beginning with a '{' are also considered to be section delimiters. The last option makes it useful for finding the beginnings of C functions. The *nroff* macros that are used for section delimiters can be adjusted. See *Sections* under *Set Commands*.

[[Move the cursor backwards to the beginning of a section.

% Move the cursor to the matching parenthesis or brace. This is very useful in C or lisp code. If the cursor is sitting on a ( ) { or }, it is moved to the matching character at the other end of the section. If the cursor is not sitting on a brace or a parenthesis, *vi* searches forward until it finds one and then

jumps to the match mate.

[cnt]H — If there is no count, move the cursor to the top left position on the screen. If there is a count, then move the cursor to the beginning of the line *cnt* lines from the top of the screen. Mnemonic: Home

[cnt]L — If there is no count, move the cursor to the beginning of the last line on the screen. If there is a count, move the cursor to the beginning of the line *cnt* lines from the bottom of the screen. Mnemonic: Last

M — Move the cursor to the beginning of the middle line on the screen. Mnemonic: Middle

m<a-z> — *Mark* the place in the file without moving the cursor; use a character '<a-z>' as the label for referring to this location in the file. See the next two commands. Mnemonic: mark Note: the *mark* command is not a motion and cannot be used as the target of commands such as *delete*.

'<a-z> — Move the cursor to the beginning of the line that is marked with the label '<a-z>'.

`<a-z> — Move the cursor to the exact position on the line that was marked with the label '<a-z>'.

'' — Move the cursor back to the beginning of the line where it was before the last *non-relative* move. A non-relative move is something such as searching or jumping to a specific line in the file, rather than moving the cursor or scrolling the screen.

`` — Move the cursor back to the exact spot on the line where it was located before the last non-relative move.

2.9.5. Searches

The following commands search for items in a file.

[cnt]f{chr} — Search forward on the line for the next or *cnt*th occurrence of the character *chr*. The cursor is placed *at* the character of interest. Mnemonic: find character

[cnt]F{chr} — Search backwards on the line for the next or *cnt*th occurrence of the character *chr*. The cursor is placed *at* the character of interest.

[cnt]t{chr} — Search forward on the line for the next or *cnt*th occurrence of the character *chr*. The cursor is placed *just preceding* the character of interest. Mnemonic: move cursor up to character

[cnt]T{chr} — Search backwards on the line for the next or *cnt*th occurrence of the character *chr*. The cursor is placed *just preceding* the character of interest.

[cnt]; — Repeat the last **f**, **F**, **t** or **T** command.

[cnt], — Repeat the last **f**, **F**, **t** or **T** command, but in the opposite search direction. This is useful if you overshoot what you are looking for.

[cnt]/[string]/<nl> — Search forward for the next occurrence of 'string'. Wraparound at the end of the file does occur. The final </> is not required.

[cnt]?[string]?<nl> — Search backwards for the next occurrence of 'string'. If a count is specified,

the count becomes the new window size. Wraparound at the beginning of the file does occur. The final <?> is not required.

n Repeat the last /[string]/ or ?[string]? search. Mnemonic: next occurrence.

N Repeat the last /[string]/ or ?[string]? search, but in the reverse direction.

:g/[string]/[editor command]<nl>
Using the : syntax, it is possible to do global searches like you can in the *ed* editor.

2.9.6. Text Insertion

The following commands insert text. Terminate all multi-character text insertions with an ESC character. You can always *undo* the last change by typing a **u**. The text insert in insertion mode can contain newlines.

a{text}<esc> Insert text immediately following the cursor position. Mnemonic: append

A{text}<esc> Insert text at the end of the current line. Mnemonic: Append

i{text}<esc> Insert text immediately preceding the cursor position. Mnemonic: insert

I{text}<esc> Insert text at the beginning of the current line.

o{text}<esc> Insert a new line after the line on which the cursor appears and insert text there. Mnemonic: open new line

O{text}<esc> Insert a new line preceding the line on which the cursor appears and insert text there.

2.9.7. Text Deletion

The following commands delete text in various ways. You can always *undo* changes by typing the **u** command.

[cnt]x Delete the character or characters starting at the cursor position.

[cnt]X Delete the character or characters starting at the character preceding the cursor position.

D Delete the remainder of the line starting at the cursor. Mnemonic: Delete the rest of line

[cnt]d{motion}
Delete one or more occurrences of the specified motion. You can use any motion from *Low Level Character Motions* and *Higher Level Text Objects* here. You can repeat the **d** (such as [*cnt*]**dd**) to delete *cnt* lines.

2.9.8. Text Replacement

Use the following commands to simultaneously delete and insert new text. You can *undo* all such actions by typing **u** following the command.

r<chr> Replace the character at the current cursor position with <chr>. This is a one-character replacement. No ESC is required for termination. Mnemonic: replace character

R{text}<esc> Start overlaying the characters on the screen with whatever you type. It does not stop until you type an ESC.

[cnt]s{text}<esc>
Substitute for *cnt* characters beginning at the current cursor position. A '$' appears at the position in the text where the *cnt*th character appears so you will know how much you are erasing. Mnemonic: substitute

[cnt]S{text}<esc>
Substitute for the entire current line or lines. If you do not give a count, a '$' appears at the end of the current line. If you give a count of more than 1, all the lines to be replaced are deleted before the insertion begins.

[cnt]c{motion}{text}<esc>
Change the specified *motion* by replacing it with the insertion text. A '$' appears at the end of the last item that is being deleted unless the deletion involves whole lines. Motion's can be any motion from sections *Low Level Character Motions* and *Higher Level Text Objects*. Repeat the **c** (such as [*cnt*]**cc**) to change *cnt* lines.

2.9.9. Moving Text

You can move chunks of text around in a number of ways with *vi*. There are nine buffers into which each piece of text which is deleted or *yanked* is put in addition to the *undo* buffer. The most recent deletion or yank is in the *undo* buffer and also usually in buffer 1, the next most recent in buffer 2, and so forth. Each new deletion pushes down all the older deletions. Deletions older than 9 disappear. There is also a set of named registers, a-z, into which text can optionally be placed. If you precede any delete or replacement type command by "<**a-z**>, that named buffer will contain the text deleted after the command is executed. For example, "**a3dd** deletes three lines starting at the current line and puts them in buffer "**a**. Referring to an upper-case letter as a buffer name (A-Z) is the same as referring to the lower-case letter, except that text placed in such a buffer is appended to it instead of replacing it. There are two more basic commands and some variations useful in getting and putting text into a file.

["<a-z>][cnt]y{motion}
Yank the specified item or *cnt* items and put in the *undo* buffer or the specified buffer. The variety of *items* that you can yank is the same as those that you can delete with the **d** command or changed with the **c** command. In the same way that **dd** means delete the current line and **cc** means replace the current line, **yy** means yank the current line.

["<a-z>][cnt]Y Yank the current line or the *cnt* lines starting from the current line. If no buffer is specified, they will go into the *undo* buffer, like any delete would. It is equivalent to **yy**. Mnemonic: **Y**ank

["<a-z>]p Put *undo* buffer or the specified buffer down *after* the cursor. If you yanked or deleted whole lines into the buffer, they are put down on the line following the line the cursor is on. If you deleted something else, like a word or sentence, it is inserted immediately following the cursor. Mnemonic: **p**ut buffer

Note that text in the named buffers remains there when you start editing a new file with the **:e** *file*CR command. Since this is so, it is possible to copy or delete text from one file and carry it over to another file in the buffers. However, the undo buffer and the ability to undo are lost when changing files.

["<a-z>]P — Put *undo* buffer or the specified buffer down *before* the cursor. If you yanked or deleted whole lines into the buffer, they are put down on the line preceding the line the cursor is on. If you deleted something else, like a word or sentence, it is inserted immediately preceding the cursor.

[cnt]>{motion} — The shift operator right shifts all the text from the line on which the cursor is located to the line where the **motion** is located. The text is shifted by one *shiftwidth*. (See *Terminal Information.*) >> means right shift the current line or lines.

[cnt]<{motion} — The shift operator left shifts all the text from the line on which the cursor is located to the line where the *item* is located. The text is shifted by one *shiftwidth*. (See *Terminal Information.*) << means left shift the current line or lines. Once the line has reached the left margin, it is not affected further.

[cnt]={motion} — Prettyprints the indicated area according to *lisp* conventions. The area should be a *lisp* s-expression.

2.9.10. Miscellaneous Commands

A number of useful miscellaneous *vi* commands follow:

ZZ — Exit from *vi*. If any changes have been made, the file is written out. Then you are returned to the shell.

^L — Redraw the current screen. This is useful if messages from a background process are displayed on the screen, if someone 'writes' to you while you are using *vi* or if for any reason garbage gets onto the screen.

^R — On dumb terminals, those not having the 'delete line' function (the vt100 for example), *vi* saves redrawing the screen when you delete a line by just marking the line with an '@' at the beginning and blanking the line. If you want to actually get rid of the lines marked with '@' and see what the page looks like, type a ^R.

. — 'Dot' repeats the last text modifying command. You can type a command once and then move to another place and repeat it by just typing '.'.

u — Undo the last command that changed the buffer. Perhaps the most important command in the editor. Mnemonic: undo

U — Undo all the text modifying commands performed on the current line since the last time you moved onto it.

[cnt]J — Join the current line and the following line. The <nl> is deleted and the two lines joined, usually with a space between the end of the first line and the beginning of what was the second line. If the first line ended with a 'period', two spaces are inserted. A count joins the next *cnt* lines. Mnemonic: Join lines

Q — Switch to *ex* editing mode. In this mode *vi* behaves very much like *ed*. The editor in this mode operates on single lines normally and does not attempt to keep the 'window' up to date. Once in this mode you can also switch to the *open* mode of editing by entering the command */line number/*open<nl>. It is similar to the normal visual mode except the window is only *one* line long. Mnemonic: Quit visual mode

^] An abbreviation for a *tag* command. The cursor should be positioned at the beginning of a word. That word is taken as a tag name, and the tag with that name is found as if it had been typed in a **:tag** command.

[cnt]![motion]{Sun cmd}<nl>
Any Sun system filter (that is, a command that reads the standard input and outputs something to the standard output) can be sent a section of the current file and have the output of the command replace the original text. Useful examples are programs like *cb*, *sort*, and *nroff*. For instance, using *sort* you can sort a section of the current file into a new list. Using !! means take a line or lines starting at the line the cursor is currently on and pass them to the Sun system command. Note: To escape to the Shell for just one command, use :![cmd]<nl> (see *High Level Commands*).

z[cnt]<nl> Reset the current window size to *cnt* lines and redraw the screen.

2.9.11. Special Insert Characters

Following are some characters that have special meanings during insert mode.

^V During inserts, typing a **^V** quotes control characters into the file. Any character typed after the **^V** is inserted into the file.

[^]^D or [0]^D **^D** without any argument backs up one *shiftwidth*. Use this to remove indentation that was inserted by the *autoindent* feature. **^^D** temporarily removes all the autoindentation, thus placing the cursor at the left margin. On the next line, the previous indent level is restored. This is useful for putting 'labels' at the left margin. **0^D** removes all autoindents and keeps it that way. Thus the cursor moves to the left margin and stays there on successive lines until you type TAB's. As with the TAB, the **^D** is effective only before you type any other 'non-autoindent' controlling characters. Mnemonic: Delete a shiftwidth

^W If the cursor is sitting on a word, **^W** moves the cursor back to the beginning of the word, erasing the word from the insert. Mnemonic: erase Word

<bs> The backspace always serves as an erase during insert modes in addition to your normal 'erase' character. To insert a <bs> into your file, quote it with the **^V**.

2.9.12. : Commands

Typing a ':' during command mode puts the cursor at the bottom on the screen in preparation for a command. In the ':' mode, you can give *vi* most *ed* commands. You can also exit from *vi* or switch to different files from this mode. Terminate all commands of this variety by a <nl>, <cr>, or ESC.

:w[!] [file] Write out the current text to the disk. It is written to the file you are editing unless you supply *file*. If *file* is supplied, the write is directed to that file instead. If that file already exists, *vi* does not write unless you use the '!' indicating you *really* want to write over the older copy of the file.

:q[!] Exit from *vi*. If you have modified the file you are currently looking at and haven't written it out, *vi* refuses to exit unless you type the '!'.

:e[!] [+ [cmd]] [file]
: Start editing a new file called *file* or start editing the current file over again. The command :e! says 'ignore the changes I've made to this file and start over from the beginning'. Use it if you really mess up the file. The optional '+ ' says instead of starting at the beginning, start at the 'end', or, if you supply *cmd*, execute *cmd* first. Use this where *cmd* is *n* (any integer) which starts at line number *n*, and */text*, which searches for 'text' and starts at the line where it is found.

^^
: Switch back to where you were before your last *tag* command. If your last *tag* command stayed within the file, ^^ returns to that tag. If you have no recent *tag* command, it returns to the same place in the previous file that it was showing when you switched to the current file.

:n[!]
: Start editing the next file in the argument list. Since you can call *vi* with multiple filenames, the :n command tells it to stop work on the current file and switch to the next file. If you have modified the current file, it has to be written out before the :n will work or else you must use '!', which discards the changes you made to the current file.

:n[!] file [file file ...]
: Replace the current argument list with a new list of files and start editing the first file in this new list.

:r file
: Read in a copy of *file* on the line after the cursor.

:r !cmd
: Execute the *cmd* and take its output and put it into the file after the current line.

:!cmd
: Execute any system Shell command.

:ta[!] tag
: *Vi* looks in the file named *tags* in the current directory. *Tags* is a file of lines in the format:

tag *filename* vi-search-command

If *vi* finds the tag you specified in the **:ta** command, it stops editing the current file if necessary. If the current file is up to date on the disk, it switches to the file specified and uses the search pattern specified to find the 'tagged' item of interest. Use this when editing multi-file C programs such as the operating system. There is a program called *ctags* which generates an appropriate *tags* file for C and f77 programs so that by saying **:ta function<nl>** you can switch to that function. It can also be useful when editing multi-file documents, though the *tags* file has to be generated manually in this case.

2.9.13. Set Commands

Vi has a number of internal variables and switches which you can set to achieve special affects. These options come in three forms, those that are switches, which toggle from off to on and back, those that require a numeric value, and those that require an alphanumeric string value. Set the toggle options by a command of the form:

:set *option*<nl>

and turn off the toggle options with the command:

:set *nooption*<nl>

To set commands requiring a value, use a command of the form:

:set *option*=*value*<nl>

To display the value of a specific option, type:

:set *option*?<nl>

To display only those that you have changed, type:

:set<nl>

and to display the long table of all the settable parameters and their current values, type:

:set all<nl>

Most of the options have a long form and an abbreviation. Both are described in the following list as well as the normal default value.

To use values other than the default every time you enter *vi*, place the appropriate *set* command in EXINIT in your environment, such as:

setenv EXINIT 'set ai aw terse sh=/bin/csh'

or

```
EXINIT='set ai aw terse sh=/bin/csh'
export EXINIT
```

for *csh* and *sh*, respectively. Place these in your *.login* or *.profile* file in your home directory.

autoindent ai — Default: noai Type: toggle
When in autoindent mode, *vi* helps you indent code by starting each line in the same column as the preceding line. Tabbing to the right with <tab> or ^T moves this boundary to the right; to move it to the left, use ^D.

autoprint ap — Default: ap Type: toggle
Displays the current line after each *ex* text modifying command. Not of much interest in the normal *vi* visual mode.

autowrite aw — Default: noaw type: toggle
Does an automatic write if there are unsaved changes before certain commands which change files or otherwise interact with the outside world are executed. These commands are **:!**, **:tag**, **:next**, **:rewind**, ^^, and ^].

beautify bf — Default: nobf Type: toggle
Discards all control characters except <tab>, <nl>, and <ff>.

directory dir — Default: dir=*/tmp* Type: string
This is the directory in which *vi* puts its temporary file.

errorbells eb — Default: noeb Type: toggle
Error messages are preceded by a <bell>.

hardtabs ht — Default: hardtabs=8 Type: numeric
This option contains the value of hardware tabs in your terminal, or of software tabs expanded by the Sun system.

ignorecase ic — Default: noic Type: toggle
Map all upper-case characters to lower case in regular expression matching.

lisp Default: nolisp Type: toggle
Autoindent for *lisp* code. The commands ( ) [[and]] are modified appropriately to affect s-expressions and functions.

list Default: nolist Type: toggle
Show the <tab> and <nl> characters visually on all displayed lines.

magic Default: magic Type: toggle
Enable the metacharacters for matching. These include . * < > **[string]** **[^string]** and **[<chr>-<chr>]**.

number nu Default: nonu Type: toggle
Display each line with its line number.

open Default: open Type: toggle
When set, prevents entering open or visual modes from *ex* or *edit*. Not of interest from *vi*.

optimize opt Default: opt Type: toggle
Useful only when using the *ex* capabilities. This option prevents automatic <cr>s from taking place, and speeds up output of indented lines, at the expense of losing typeahead on some versions of the operating system.

paragraphs para Default: para=IPLPPPQPP bp Type: string
Each pair of characters in the string indicates *nroff* macros which are to be treated as the beginning of a paragraph for the { and } commands. The default string is for the *-ms*. To indicate one letter *nroff* macros, such as '.P' or '.H', quote a space in for the second character position. For example:

:set paragraphs=PP\ bp<nl>

causes *vi* to consider '.PP' and '.bp' as paragraph delimiters.

prompt Default: prompt Type: toggle
In *ex* command mode the prompt character : is displayed when *ex* is waiting for a command. This is not of interest from *vi*.

redraw Default: noredraw Type: toggle
On dumb terminals, force the screen to always be up to date by sending great amounts of output. Useful only at high speeds.

report Default: report=5 Type: numeric
Set the threshold for the number of lines modified. When more than this number of lines is modified, removed, or yanked, *vi* reports the number of lines changed at the bottom of the screen.

scroll Default: scroll={1/2 window} Type: numeric
This is the number of lines that the screen scrolls up or down when using the ^U and ^D commands.

sections Default: sections=SHNHH HU Type: string
Each two-character pair of this string specifies *nroff* macro names, which are to be treated as the beginning of a section by the]] and [[commands. The default string is for the *-ms* and *-mm* macros. To enter one-letter *nroff* macros, use a quoted space as the second character. See *Paragraphs* for a fuller explanation.

shell sh Default: sh=from environment SHELL or /bin/sh Type: string
Specify the name of the *sh* to be used for 'escaped' commands.

shiftwidth sw
Default: sw=8 Type: numeric
Specify the number of spaces that a ^T or ^D will move over for indenting, and the amount that < and > will shift by.

showmatch sm
Default: nosm Type: toggle
When a) or } is typed, show the matching (or { by moving the cursor to it for one second if it is on the current screen.

slowopen slow
Default: terminal dependent Type: toggle
Prevent updating the screen some of the time to improve speed on terminals that are slow and dumb.

tabstop ts
Default: ts=8 Type: numeric
<tab>s are expanded to boundaries that are multiples of this value.

taglength tl
Default: tl=0 Type: numeric
If nonzero, tag names are only significant to this many characters.

term
Default: (from environment **TERM**, else dumb) Type: string
This is the terminal and controls the visual displays. It cannot be changed when in visual mode; you have to type a Q to change to command mode, type a *set term* command, and enter *vi* to get back into visual. Or exit from *vi*, fix $TERM, and re-enter. The definitions that drive a particular terminal type are in the file */etc/termcap*.

terse
Default: terse Type: toggle
When set, the error diagnostics are short.

warn
Default: warn Type: toggle
Warns if you try to escape to the Shell without writing out the current changes.

window
Default: window={8 at 600 baud or less, 16 at 1200 baud, and screen size – 1 at 2400 baud or more} Type: numeric
Specify the number of lines in the window whenever *vi* must redraw an entire screen. It is useful to make this size smaller if you are on a slow line.

w300, w1200, w9600
Set the window, but only within the corresponding speed ranges. They are useful in an EXINIT to fine tune window sizes. For example,

```
set w300=4 w1200=12
```

produces a 4-line window at speeds up to 600 baud, a 12-line window at 1200 baud, and a full-screen window (the default) at over 1200 baud.

wrapscan ws
Default: ws Type: toggle
Searches will wrap around the end of the file when is option is set. When it is off, the search will terminate when it reaches the end or the beginning of the file.

wrapmargin wm
Default: wm=0 Type: numeric
Vi automatically inserts a <nl> when it finds a natural break point (usually a <sp> between words) that occurs within *wm* spaces of the right margin. Therefore with 'wm=0', the option is off. Setting it to 10 means that any time you are within 10 spaces of the right margin, *vi* looks for a <sp> or <tab> which it can replace with a <nl>. This is convenient if you forget to look at the screen while you type. If you go past the margin (even in the middle of a word), the entire word is erased and rewritten on the next line.

writeany wa Default: nowa Type: toggle
Vi normally makes a number of checks before it writes out a file. This prevents you from inadvertently destroying a file. When the *writeany* option is enabled, *vi* no longer makes these checks.

2.9.14. Character Functions

This section describes how the editor uses each character. The characters are presented in their order in the ASCII character set: control characters come first, then most special characters, the digits, upper and finally lower-case characters.

For each character we give the meaning it has as a command and any meaning it has during insert mode. It may only have meaning as a command.

^@ Not a command character. If typed as the first character of an insertion, it is replaced with the last text inserted, and the insert terminates. Only 128 characters are saved from the last insert; if more characters were inserted the mechanism is not available. A ^@ cannot be part of the file due to the editor implementation.

^A Unused.

^B Backward window. A count specifies repetition. Two lines of continuity are kept if possible.

^C Unused.

^D As a command, scrolls down a half window of text. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands. During an insert, backtabs over *autoindent* white space at the beginning of a line. This white space cannot be backspaced over.

^E Exposes one more line below the current screen in the file, leaving the cursor where it is if possible.

^F Move forward one window. A count specifies repetition. Two lines of continuity are kept if possible.

^G Equivalent to :f CR, displaying the current file, whether it has been modified, the current line number and the number of lines in the file, and the percentage of the way through the file that you are.

^H (BS) Same as ← (see **h**). During an insert, eliminates the last input character, backing over it but not erasing it; it remains so you can see what you typed if you wish to type something only slightly different.

^I (TAB) Not a command character. When inserted it prints as some number of spaces. When the cursor is at a tab character, it rests at the last of the spaces which represent the tab. The *tabstop* option controls the spacing of tabstops.

^J (LF) Same as ↓ (see **j**).

^K Unused.

^L The ASCII formfeed character, which clears and redraws the screen. This is useful after a transmission error, if characters typed by a program other than the editor scramble the screen, or after output is stopped by an interrupt.

^M (CR) A carriage return advances to the next line, at the first non-white position in the line. Given a count, it advances that many lines. During an insert, a CR causes the insert to continue onto another line.

^N	Same as ↓ (see **j**).
^O	Unused.
^P	Same as ^ (see **k**).
^Q	Not a command character. In input mode, ^Q quotes the next character, the same as ^V, except that some teletype drivers will eat the ^Q so that *vi* never sees it. Resumes operation suspended by ^S.
^R	Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines with only a single @ character on them). On hardcopy terminals in *open* mode, retypes the current line.
^S	Some teletype drivers use ^S to suspend output until ^Q is pressed. Unused.
^T	Not a command character. During an insert with *autoindent* set and at the beginning of the line, inserts *shiftwidth* white space.
^U	Scrolls the screen up, inverting ^D which scrolls down. Counts work as they do for ^D, and the previous scroll amount is common to both. On a dumb terminal, ^U will often necessitate clearing and redrawing the screen further back in the file.
^V	Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.
^W	Not a command character. During an insert, backs up as **b** does in command mode; the deleted characters remain on the display (see ^H).
^X	Unused.
^Y	Exposes one more line above the current screen, leaving the cursor where it is if possible. (No mnemonic value for this key; however, it is next to ^U which scrolls up.)
^Z	Stops the editor, exiting to the top level Shell. Same as **:stop**CR.
^[(ESC)	Cancels a partially formed command, such as a **s** when no following character has yet been given; terminates inputs on the last line (read by commands such as **:** **/** and **?**); ends insertions of new text into the buffer. If an ESC is given when quiescent in command state, the editor flashes the screen or rings the bell. You can thus type ESC if you don't know what is happening till the editor flashes the screen. If you don't know if you are in insert mode, you can type ESC**a**, and then material to be input; the material is inserted correctly whether or not you were in insert mode when you started.
^\	Unused.
^]	Searches for the word which is after the cursor as a tag. Equivalent to typing **:ta**, this word, and then a CR. Mnemonically, this command is 'go right to'.
^^	Equivalent to **:e #**CR, returning to the previous position in the last edited file, or editing a file which you specified if you got a 'No write since last change diagnostic' and do not want to have to type the filename again. You have to do a **:w** before ^^ will work in this case. If you do not wish to write the file you should do **:e!** #CR instead.
^_	Unused. Reserved as the command character for the Tektronix 4025 and 4027 terminal.
SPACE	Same as → (see **l**).

! An operator, which processes lines from the buffer with reformatting commands. Follow ! with the object to be processed, and then the command name terminated by CR. Doubling ! and preceding it by a count filters the count lines; otherwise the count is passed on to the object after the !. Thus **2!}***fmt*CR reformats the next two paragraphs by running them through the program *fmt*. If you are working on LISP, the command **!%***grind*CR, given at the beginning of a function, will run the text of the function through the LISP grinder. (*do we support grind?*) (The *grind* command and may not be present at all installations.) To read a file or the output of a command into the buffer, use **:r**. To simply execute a command, use **:!**.

" Precedes a named buffer specification. There are named buffers **1–9** used for saving deleted text and named buffers **a–z** into which you can place text.

The macro character which, when followed by a number, will substitute for a function key on terminals without function keys. In input mode, if this is your erase character, it will delete the last character you typed in input mode, and must be preceded with a \ to insert it, since it normally backs over the last input character you gave.

$ Moves to the end of the current line. If you **:se list**CR, the end of each line is indicated by showing a $ after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus **2$** advances to the end of the following line.

% Moves to the parenthesis or brace { } which balances the parenthesis or brace at the current cursor position.

& A synonym for **:&**CR, by analogy with the *ex* **&** command.

' When followed by a '', returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a–z**, returns to the line which was marked with this letter with a **m** command, at the first non-white character in the line. When used with an operator such as **d**, the operation takes place over complete lines; if you use **`**, the operation takes place from the exact marked place to the current cursor position within the line.

(Retreats to the beginning of a sentence, or to the beginning of a LISP s-expression if the *lisp* option is set. A sentence ends at a **.** **!** or **?** and is followed by either the end of a line or by two spaces. Any number of closing **)** **]** **"** and **'** characters may appear after the **.** **!** or **?**, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see **{** and **[[** below). A count advances that many sentences.

) Advances to the beginning of a sentence. A count repeats the effect. See **(** above for the definition of a sentence.

***** Unused.

+ Same as CR when used as a command.

, Reverse of the last **f** **F** **t** or **T** command, looking the other way in the current line. Especially useful after typing too many **;** characters. A count repeats the search.

– Retreats to the previous line at the first non-white character. This is the inverse of **+** and RETURN. If the line moved to is not on the screen, the screen is scrolled, or cleared and redrawn if scrolling is not possible. If a large amount

of scrolling is required, the screen is also cleared and redrawn, with the current line at the center.

. Repeats the last command which changed the buffer. Especially useful when deleting words or lines; you can delete some words or lines and then type . to delete more words or lines. Given a count, it passes it on to the command being repeated. Thus after a **2dw**, **3.** deletes three words.

/ Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during the input on the bottom line; an ESC returns to command state without ever searching. The search begins when you type CR to terminate the pattern; the cursor moves to the beginning of the last line to indicate that the search is in progress; you can then terminate the search with a ^C (orDEL or RUB), or by backspacing when at the beginning of the bottom line, returning the cursor to its initial position. Searches normally wrap end-around to find a string anywhere in the buffer.

When used with an operator, the enclosed region is normally affected. By mentioning an offset from the line matched by the pattern, you can affect whole lines. To do this, give a pattern with a closing / and then an offset $+n$ or $-n$.

To include the character / in the search string, you must escape it with a preceding \. A ^ at the beginning of the pattern forces the match to occur at the beginning of a line only; this may speed the search. A $ at the end of the pattern forces the match to occur at the end of a line only. More extended pattern matching is available. Unless you set **nomagic** in your *.login* file (*?*), you will have to precede the characters . [* and ~ in the search pattern with a \ to get them to work as you would naively expect.

0 Moves to the first character on the current line. Also used, in forming numbers, after an initial **1–9**.

1–9 Used to form numeric arguments to commands.

: A prefix to a set of commands for file and option manipulation and escapes to the system. Input is given on the bottom line and terminated with a CR, and the command is then executed. You can return to where you were by typing ESC or DEL if you type : accidentally.

; Repeats the last single character find which used **f F t** or **T**. A count iterates the basic scan.

< An operator which shifts lines left one *shiftwidth*, normally 8 spaces. Like all operators, affects lines when repeated, as in **<<**. Counts are passed through to the basic object, thus **3<<** shifts three lines.

= Reindents line for LISP, as though they were typed in with *lisp* and *autoindent* set.

> An operator which shifts lines right one *shiftwidth*, normally 8 spaces. Affects lines when repeated as in **>>**. Counts repeat the basic object.

? Scans backwards, the opposite of **/**. See the **/** description above for details on scanning.

@ A macro character. If this is your kill character, you must escape it with a \ to type it in during input mode, as it normally backs over the input you have given on the current line.

A Appends at the end of line; a synonym for **$a**.

B Backs up a word, where words are composed of non-blank sequences, placing the cursor at the beginning of the word. A count repeats the effect.

C Changes the rest of the text on the current line; a synonym for **c$**.

D Deletes the rest of the text on the current line; a synonym for **d$**.

E Moves forward to the end of a word, defined as blanks and non-blanks, like **B** and **W**. A count repeats the effect.

F Finds a single following character backwards in the current line. A count repeats this search that many times.

G Goes to the line number given as preceding argument, or to the end of the file if you do not give a preceding count. The screen is redrawn with the new current line in the center if necessary.

H Home arrow. Homes the cursor to the top line on the screen. If a count is given, the cursor is moved to the count'th line on the screen. In any case the cursor is moved to the first non-white character on the line. If used as the target of an operator, full lines are affected.

I Inserts at the beginning of a line; a synonym for **^i**.

J Joins together lines, supplying appropriate white space: one space between words, two spaces after a '.', and no spaces at all if the first character of the joined on line is). A count causes that many lines to be joined rather than the default two.

K Unused.

L Moves the cursor to the first non-white character of the last line on the screen. With a count, to the first non-white of the count'th line from the bottom. Operators affect whole lines when used with **L**.

M Moves the cursor to the middle line on the screen, at the first non-white position on the line.

N Scans for the next match of the last pattern given to **/** or **?**, but in the reverse direction; this is the reverse of **n**.

O Opens a new line above the current line and inputs text there up to an ESC. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the *slowopen* option works better.

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted between the characters before and at the cursor. May be preceded by a named buffer specification **"*x*** to retrieve the contents of the buffer; buffers **1–9** contain deleted material, buffers **a–z** are available for general use.

Q Quits from *vi* to *ex* command mode. In this mode, whole lines form commands, ending with a RETURN. You can give all the **:** commands; the editor supplies the **:** as a prompt.

R Replaces characters on the screen with characters you type (overlay fashion). Terminates with an ESC.

S Changes whole lines, a synonym for **cc**. A count substitutes for that many lines. The lines are saved in the numeric buffers, and erased on the screen before the substitution begins.

T Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect. Most useful with operators such as **d**.

U Restores the current line to its state before you started changing it.

V Unused.

W Moves forward to the beginning of a word in the current line, where words are defined as sequences of blank/non-blank characters. A count repeats the effect.

X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y Yanks a copy of the current line into the unnamed buffer, to be put back by a later **p** or **P**; a very useful synonym for **yy**. A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.

ZZ Exits the editor. (Same as **:x**CR.) If any changes have been made, the buffer is written out to the current file. Then the editor quits.

[[Backs up to the previous section boundary. A section begins at each macro in the *sections* option, normally a '.NH' or '.SH' and also at lines which which start with a formfeed ^**L**. Lines beginning with { also stop **[[**; this makes it useful for looking backwards, a function at a time, in C programs. If the *lisp* option is set, stops at each (at the beginning of a line, and is thus useful for moving backwards at the top level LISP objects.

**** Unused.

]] Forward to a section boundary; see **[[** for a definition.

^ Moves to the first non-white position on the current line.

_ Unused.

`** When followed by a ` returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter **a–**z**, returns to the position which was marked with this letter with an **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; if you use ', the operation takes place over complete lines.

a Appends arbitrary text after the current cursor position; the insert can continue onto multiple lines by using RETURN within the insert. A count causes the inserted text to be replicated, but only if the inserted text is all on one line. Terminate the insertion with an ESC.

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect.

c An operator which changes the following object, replacing it with the following input text up to an ESC. If more than part of a single line is affected, the text which is changed is saved in the numeric named buffers. If only part of the current line is affected, the last character to be changed away is marked with a **$**. A count causes that many objects to be affected, thus both **3c)** and **c3)** change the following three sentences.

d An operator which deletes the following object. If more than part of a line is affected, the text is saved in the numeric buffers. A count causes that many objects to be affected; thus **3dw** is the same as **d3w**.

e Advances to the end of the next word, defined as for **b** and **w**. A count repeats the effect.

f Finds the first instance of the next character following the cursor on the current line. A count repeats the find.

g Unused.

Arrow keys **h**, **j**, **k**, **l**, and **H**.

h Left arrow. Moves the cursor one character to the left. Like the other arrow keys, either **h**, the left arrow key, or one of the synonyms (^**H**) has the same effect. A count repeats the effect.

i Inserts text before the cursor, otherwise like **a**.

j Down arrow. Moves the cursor one line down in the same column. If the position does not exist, *vi* comes as close as possible to the same column. Synonyms include ^**J** (linefeed) and ^**N**.

k Up arrow. Moves the cursor one line up. ^**P** is a synonym.

l Right arrow. Moves the cursor one character to the right. SPACE is a synonym.

m Marks the current position of the cursor in the mark register which is specified by the next character **a**–**z**. Return to this position or use with an operator using ‘‘ or ‘’.

n Repeats the last **/** or **?** scanning commands.

o Opens new lines below the current line; otherwise like **O**.

p Puts text after/below the cursor; otherwise like **P**.

q Unused.

r Replaces the single character at the cursor with a single character you type. The new character may be a RETURN; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given; see **R** above which is the more usually useful iteration of **r**.

s Changes the single character under the cursor to the text which follows up to an ESC; given a count, that many characters from the current line are changed. The last character to be changed is marked with **$** as in **c**.

t Advances the cursor up to the character before the next character typed. Most useful with operators such as **d** and **c** to delete the characters up to a following character. You can use **.** to delete more if this doesn't delete enough the first time.

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers.

v Unused.

w Advances to the beginning of the next word, as defined by **b**.

x Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.

y An operator, yanks the following object into the unnamed temporary buffer. If preceded by a named buffer specification, "*x*, the text is placed in that buffer

also. Text can be recovered by a later p or P.

z Redraws the screen with the current line placed as specified by the following character: RETURN specifies the top of the screen, . the center of the screen, and '–' at the bottom of the screen. A count before the z gives the number of the line to place in the center of the screen instead of the default current line. To change the window size, use a count after the z and before the RETURN, as in z5<CR>.

{ Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the *paragraphs* option, normally '.IP', '.LP', '.PP', '.QP' and '.bp'. A paragraph also begins after a completely empty line, and at each section boundary (see [[above).

| Places the cursor on the character in the column specified by the count.

} Advances to the beginning of the next paragraph. See { for the definition of paragraph.

~ Unused.

^C (DEL) Interrupts the editor, returning it to command accepting state.

2.10. Terminal Information

Vi works on a large number of display terminals. You can edit a terminal description file to drive new terminals. While it is advantageous to have an intelligent terminal which can locally insert and delete lines and characters from the display, *vi* functions quite well on dumb terminals over slow phone lines. *Vi* allows for the low bandwidth in these situations and uses smaller window sizes and different display updating algorithms to make best use of the limited speed available.

You can also use the *vi* command set on hardcopy terminals, storage tubes and 'glass tty's' using a one-line editing window.

2.10.1. Specifying Terminal Type

Before you can start *vi* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Table 2-6: Terminal Types

Code	Full Name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose for example that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system your terminal type:

```
logo% setenv TERM 2621
```

If you are using the Bourne Shell, use:

```
$ TERM=2621
$ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, use the *tset* program. If you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use *csh*) is

```
setenv TERM `tset - -d mime`
```

or for your *.profile* file (if you use *sh*):

```
TERM=`tset - -d mime`
```

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. You can use *tset* to change the erase and kill characters too.

2.10.2. Special Arrangements for Startup

Vi takes the value of **$TERM** and looks up the characteristics of that terminal in the file */etc/termcap*. If you don't know *vi*'s name for the terminal you are working on, look in */etc/termcap*. The editor adopts the convention that a null string in the environment is the same as not being set. This applies to TERM, TERMCAP, and EXINIT.

When *vi* starts, it attempts to read the variable EXINIT from your environment. If that exists, it takes the values in it as the default values for certain of its internal constants. See *Set Values* for further details. If EXINIT doesn't exist, you will get all the normal defaults.

Should you inadvertently hang up the phone while inside *vi*, or should something else go wrong, all may not be lost. Upon returning to the system, type:

logo% **vi -r** *file*

This will normally recover the file. If there is more than one temporary file for a specific filename, *vi* recovers the newest one. You can get an older version by recovering the file more than once. The command **vi -r** without a filename lists the files from an on-line list that were saved in the last system crash (but *not* the file just saved when the phone was hung up).

2.10.3. Open Mode on Hardcopy Terminals and 'Glass tty's'

If you are on a hardcopy terminal or a terminal which does not have a cursor which can move off the bottom line, you can still use the command set of *vi*, but in a different mode. When you give a *vi* command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in *ex*, which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is displayed. In *open* mode the editor uses a single-line window into the file, and moving backward and forward in the file displays new lines, always below the current line. Two *vi* commands that work differently in *open* are: **z** and **^R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the **^R** command retypes the current line. On such terminals, *vi* normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters which are deleted. It also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals which can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

2.10.4. Editing on Slow Terminals

When you are on a slow terminal, it is important to limit the amount of output which is generated to your screen so that you will not suffer long delays, waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command:

:se slowCR

If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by:

:se noslowCR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command:

:se redrawCR

This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command:

:se noredrawCR.

The editor also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window which is redrawn each time the screen is cleared by giving window size as an argument to the commands which cause large screen motions:

: / ? [[ ]] ` '

Thus if you are searching for a particular instance of a common string in a file, you can precede the first search command by a small number, say 3, and the editor will draw three line windows around each instance of the string which it locates.

You can expand or contract the window size, placing the current line as you choose, with the **z** command, as in **z5**<CR>, which changes the window to five lines. You can also use **.** or **–**. Thus the command **z5.** redraws the screen with the current line in the center of a five-line window. Note that the command **5z.** has an entirely different effect, placing line 5 in the center of a new window. Use **–**, as in **5z–** to position the cursor at line 5 in the file.

The default window sizes are 8 lines at 300 baud, 16 lines at 1200 baud, usually also 16 for a typical 24 line CRT). and full screen size at 9600 baud. Any baud rate less than 1200 behaves like 300, and any over 1200 like 9600.

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by typing a DEL or RUB as usual. If you do this, you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by typing a **^L**; or move or search again, ignoring the current state of the display.

See the section on *open* mode for another way to use the *vi* command set on slow terminals.

2.10.5. Upper-case Only Terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters which you normally type are converted to lower case, and you can type upper-case letters by preceding them with a '\'. The characters { ~ } | ` are not available on such terminals, but you can escape them as \(\^ \) \! \'. These characters are represented on the display in the same way they are typed.[9]

[9] The '\' character you give will not echo until you type another key.

2.11. Command Summary

The following is a quick summary of frequently used commands. Refer to the quick reference pages for a reference summary of all commands.

Table 2-7: Frequently Used Commands

SPACE	advance the cursor one position
^B	backwards to previous page
^D	scrolls down in the file
^E	exposes another line at the bottom
^F	forward to next page
^G	tell what is going on
^H	backspace the cursor
^N	next line, same column
^P	previous line, same column
^U	scrolls up in the file
^Y	exposes another line at the top (v3)
+	next line, at the beginning
–	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or ? pattern
w	word after this word
^W	erase a word during an insert
DEL	your erase (or ^H), erases a character during an insert
^U	your kill (or ^X), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text
d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change
^	first non-white on line
$	end of line
)	forward sentence

}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

Vi Quick Reference

Entering/Leaving vi

% vi *name*	edit *name* at top
% vi +*n name*	... at line *n*
% vi + *name*	... at end
% vi -r	list saved files
% vi -r *name*	recover file *name*
% vi *name* ...	edit first; rest via :n
% vi -t *tag*	start at *tag*
% vi +/*pat name*	search for *pat*
% view *name*	read only mode
ZZ	exit from *vi*, saving changes
^Z	stop *vi* for later resumption

The Display

Last line	Error messages, echoing input to : / ? and !, feedback about i/o and large changes.
@ lines	On screen only, not in file.
~ lines	Lines past end of file.
^*x*	Control characters, DEL is delete.
tabs	Expand to spaces, cursor at last.

Vi Modes

Command	Normal and initial state. Others return here. ESC (escape) cancels partial command.
Insert	Entered by a i A I o O c C s S R. Arbitrary text then terminates with ESC character, or abnormally with interrupt.
Last line	Reading input for : / ? or !; terminate with ESC or CR to execute, interrupt to cancel.

Counts Before vi Commands

line/column number	z G |
scroll amount	^D ^U
replicate insert	a i A I
repeat effect	most rest

Simple Commands

dw	delete a word
de	... leaving punctuation
dd	delete a line
3dd	... 3 lines
i*text*ESC	insert text *abc*
cw*new*ESC	change word to *new*
ea*s*ESC	pluralize word
xp	transpose characters

Interrupting, Cancelling

ESC	end insert or incomplete cmd
^C	interrupt (or DEL)
^L	refresh screen if scrambled

File Manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e *name*	edit file *name*
:e!	reedit, discard changes
:e + *name*	edit, starting at end
:e +*n*	edit starting at line *n*
:e #	edit alternate file
^^	synonym for :e #
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:sh	run shell, then return
:!*cmd*	run *cmd*, then return
:n	edit next file in arglist
:n *args*	specify new arglist
:f	show current file and line
^G	synonym for :f
:ta *tag*	to tag file entry *tag*
^]	:ta, following word is *tag*

Positioning within File

^F	forward screenfull
^B	backward screenfull
^D	scroll down half screen
^U	scroll up half screen
G	goto line (end default)
/*pat*	next line matching *pat*
?*pat*	prev line matching *pat*
n	repeat last / or ?
N	reverse last / or ?
/*pat*/+*n*	n'th line after *pat*
?*pat*?-*n*	n'th line before *pat*
]]	next section/function
[[	previous section/function
%	find matching ( ) { or }

Adjusting the Screen

^L	clear and redraw
^R	retype, eliminate @ lines
zCR	redraw, current at window top
z-	... at bottom
z.	... at center
/*pat*/z-	*pat* line at bottom
z*n*.	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and Returning

``	previous context
''	... at first non-white in line
m*x*	mark position with letter *x*
`*x*	to mark *x*
'*x*	... at first non-white in line

Line Positioning

H	home window line
L	last window line
M	middle window line
+	next line, at first non-white
–	previous line, at first non-white
CR	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

Character Positioning

^	first non white
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^H	same as ←
space	same as →
f*x*	find *x* forward
F*x*	f backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last f F t or T
,	inverse of ;
|	to specified column
%	find matching ({) or }

Words, Sentences, Paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back W
E	to end of W

Commands for LISP

)	Forward s-expression
}	... but don't stop at atoms
(	Back s-expression
{	... but don't stop at atoms

Corrections During Insert

^H	erase last character
^W	erases last word
erase	your erase, same as ^H
kill	your kill, erase input this line
\	escapes ^H, your erase and kill
ESC	ends insertion, back to command
^C	interrupt, terminates insert
^D	backtab over *autoindent*
^^D	kill *autoindent*, save for next
0^D	... but at margin next also
^V	quote non-printing character

Insert and Replace

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous Operations

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

Yank and Put

p	put back lines
P	put before
"*x*p	put from buffer *x*
"*x*y	yank to buffer *x*
"*x*d	delete into buffer *x*

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d*p	retrieve *d*'th last delete

Ex Quick Reference

Entering/Leaving ex

% ex *name*	edit *name*, start at end
% ex +*n* *name*	... at line *n*
% ex –t *tag*	start at *tag*
% ex –r	list saved files
% ex –r *name*	recover file *name*
% ex *name* ...	edit first; rest via :n
% ex –R *name*	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

ex States

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by **a** **i** and **c**. Arbitrary text then terminates with line having only . character on it or abnormally with interrupt.
Open/visual	Entered by **open** or **vi**, terminates with Q or ^\.

ex Commands

abbrev	ab	next	n	unabbrev	una
append	a	number	nu	undo	u
args	ar	open	o	unmap	unm
change	c	preserve	pre	version	ve
copy	co	print	p	visual	vi
delete	d	put	pu	write	w
edit	e	quit	q	xit	x
file	f	read	re	yank	ya
global	g	recover	rec	window	z
insert	i	rewind	rew	escape	!
join	j	set	se	shift	<
list	l	shell	sh	print next	CR
map		source	so	resubst	&
mark	ma	stop	st	rshift	>
move	m	substitute	s	scroll	^D

ex Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
–	previous	*'x*	marked with *x*
+*n*	*n* forward	''	previous context
%	1,$		

Specifying Terminal Type

% setenv TERM *type*	(for *csh*)
$ TERM=*type*; export TERM	(for *sh*)

See also *tset* in the user's manual.

Some Terminal Types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

Initializing Options

EXINIT	place **set**'s here in environment var.
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Useful Options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and } as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer
wrapmargin	wm	automatic line splitting

Scanning Pattern Formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*
[↑*str*]	... not in *str*
[*x*–*y*]	... between *x* and *y*
*	any number of preceding

Table of Contents

Chapter 3

Command Reference for the ex Line Editor

This chapter[1] provides reference material for *ex*, the line-oriented text editor, which also supports display oriented editing in the form of the *vi* editor described in *Using Vi, the Visual Display Editor*. The contents of this chapter describe the line-oriented part of *ex*. You can also use these commands with *vi*. For a summary of *ex* commands, see the *Ex Quick Reference*

3.1. Using ex

Ex has a set of options, which you can use to tailor *ex* 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, and we assume that you are running *ex* on a Sun Workstation.

If there is a variable EXINIT in the environment, *ex* executes 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* are executed before each editor session.

If you are running *ex* on a terminal, *ex* determines the terminal type from the TERM variable in the environment when invoked. It there is a TERMCAP variable in the environment, and the type of the terminal described there matches the TERM variable, that description is used. Also if the TERMCAP variable contains a pathname (beginning with a **/**),*ex* seeks the description of the terminal in that file, rather than in the default */etc/termcap*.)

The standard *ex* command format follows. Brackets '[' ']' surround optional parameters here.

logo% **ex** [**–**] [**–v**] [**–t** *tag*] [**–r**] [**–l**] [**–w***n*] [**–x**] [**–R**] [**+***command*] *filename* ...

The most common case edits a single file with no options, that is,:

logo% **ex** *filename*

The '–' command line option option suppresses all interactive-user feedback and is useful in processing *ex* 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.

Use the **–r** option to recover a file after an editor or system problem, retrieving the last saved version of the named file or, if no file is specified, displaying a list of saved files. The **–l** option sets up for editing LISP, setting the *showmatch* and *lisp* options. 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* in *User's Manual for the Sun*

[1] The material in this chapter is derived from *Ex Reference Manual*, W.N. Joy, M. Horton, University of California, Berkeley.

Workstation). The **–R** option sets the *readonly* option at the start. If set, writes will fail unless you use an '!' after the write. This option affect **ZZ**, *autowrite* and anything that writes to guarantee you won't clobber a file by accident. *Filename* 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, it defaults to '$', initially positioning *ex* at the last line of the first file. Other useful commands here are scanning patterns of the form '/*pat*' or line numbers, such as '+ 100', which means 'start at line 100.'

3.2. File Manipulation

The following describes commands for handling files.

3.2.1. Current File

Ex normally edits the contents of a single file, whose name is recorded in the *current* filename. *Ex* performs all editing actions in a buffer 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 you write the buffer contents 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 filename, 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 filename, 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*, *ex* will not normally write on it if it already exists. The *file* command will say '[Not edited]' if the current file is not considered edited.

3.2.2. Alternate File

Each time a new value is given to the current filename, the previous current filename is saved as the *alternate* filename. Similarly if a file is mentioned but does not become the current file, it is saved as the alternate filename.

3.2.3. Filename Expansion

You may specify filenames within the editor using the normal Shell expansion conventions. In addition, the character '%' in filenames is replaced by the *current* filename and the character '#' by the *alternate* filename. 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.

3.3. Special Characters

Some characters take on special meanings when used in context searches and in patterns given to the *substitute* command. For *edit*, these are '^' and '$', meaning the beginning and end of a line, respectively. *Ex* has the following additional special characters:

. & * [ ] ~

To use one of the special characters as its simple graphic representation rather than with its special meaning, precede it by a backslash (\). The backslash always has a special meaning.

3.3.1. Multiple Files and Named Buffers

If more than one file is given on the *ex* command line, the first file is edited as described above. The remaining arguments are placed with the first file in the *argument list*. You can display the current argument list with the *args* command. To edit the next file in the argument list, use the *next* command. You may also respecify the argument list 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.

To save 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*. 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.

3.3.2. Read Only Mode

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 with **:w** *newfilename*, or can use the **:w!** form of write, even while in read only mode.

3.4. Exceptional Conditions

The following describes additional editing situations.

3.4.1. Errors and Interrupts

When errors occur *ex* flashes the workstation screen and displays an error diagnostic. If the primary input is from a file, editor processing terminates. If you interrupt *ex*, it displays 'Interrupt' and returns to its command level. If the primary input is a file, *ex* exits when this occurs.

3.4.2. Recovering If Something Goes Wrong

If something goes wrong and the buffer has been modified since it was last written out, or if the system crashes, either the editor or the system (after it reboots) attempts 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 problem. To recover a file, use the **–r** option. If you were editing the file *resume* for example, change to the directory where you were

when the problem occurred, and use *ex* with the **–r** (recover) option:

logo% **ex –r** *file*

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 the system has gone down. Use the **–r** option without a following filename:

logo% **ex –r**

to display 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.

3.5. Editing Modes

Ex has five distinct modes. The primary mode is *command* mode. You type in commands in command mode when a ':' prompt is present, and execute them each time you send a complete line. In *insert* mode, *ex* gathers input lines and places them in the file. The *append, insert,* and *change* commands use insert mode. No prompt is displayed when you are in text input mode. To leave this mode and return to command mode, type a '.' alone at the beginning of a line.

The last three modes are *open* and *visual* modes, entered by the commands of the same names, and, within open and visual modes *text insertion* mode. In *open* and *visual* modes, you do local editing operations on the text in the file. The *open* command displays one line at a time on the screen, while *visual* works on the workstation and CRT terminals with random positioning cursors, using the screen as a single window for file editing changes. See *Using Vi, The Visual Display Editor* for descriptions of these modes.

3.6. Command Structure

Most command names are English words; you can use initial prefixes of the words as acceptable abbreviations. The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the command *substitute* can be abbreviated as 's' while the shortest available abbreviation for the *set* command is **se**. See *Command Reference* for descriptions and acceptable abbreviations.

3.6.1. Specifying 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. Counts are rounded down if necessary. Thus the command **10p** displays the tenth line in the buffer, while **d5** deletes five lines from the buffer, starting with the current line.

Some commands take other information or parameters, that you provide after the command name. Examples would be option names in a *set* command such as, **set number**, a filename in an *edit* command, a regular expression in a *substitute* command, or a target address for a *copy* command, such as, **1,5 copy 25**.

3.6.2. Invoking 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. You can control some of the default variants with options; in this case, the '!' serves to toggle the default.

3.6.3. Flags after Commands

You may place the characters '#', **p** and **l** after many commands. You must precede a **p** or **l** by a blank or tab except in the single special case of **dp**. The command that these characters abbreviates is executed after the command completes. Since *ex* normally shows the new current line after each change, **p** is rarely necessary. You can also give any number of '+' or '–' characters with these flags. If they appear, the specified offset is applied to the current line value before the display command is executed.

3.6.4. Writing Comments

It is possible to give editor commands which are ignored. This is useful when making complex editor scripts for which comments are desired. Use the double quote ' " ' as the comment character. Any command line beginning with ' " ' is ignored. You can also put comments beginning with ' " ' at the ends of commands, except in cases where they could be confused as part of text, for example as Shell escapes and the *substitute* and *map* commands.

3.6.5. Putting Multiple Commands on a Line

You can place more than one *ex* command 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 '|'.

3.6.6. 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 you may quickly and easily reverse them with *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.

3.7. Command Addressing

The following describes the editor commands called *addressing primitives*.

3.7.1. Addressing Primitives

. The current line. The current line is traditionally called 'dot' because you address it with a dot '.'. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, so you rarely use

'.' 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. The forms '.+ 3' '+ 3' and '+ + + ' are all equivalent; if the current line is line 100, they all address line 103.

/pat/ ?pat?
Scan forward and backward respectively for a line containing *pat*, a regular expression (as defined below in *Regular Expressions and Substitute Replacement Patterns*. The scans normally wrap around the end of the buffer. If all that is desired is to show the next line containing *pat*, you may omit trailing / or ?. If you omit *pat* or leave it explicitly empty, the last regular expression specified is located. 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.

'' *'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. You can also establish marks with the *mark* command, using single lower-case letters *x* and the marked lines referred to as *'x* '.

3.7.2. 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 you give more addresses than the command requires, 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. Null address specifications are permitted in a list of addresses; the default in this case is the current line '.'. So ',100' is equivalent to '.,100'. It is an error to give a prefix address to a command which expects none.

3.8. Regular Expressions and Substitute Replacement Patterns

3.8.1. 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*, that is '//' or '??'.

3.8.2. 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, namely '^' (beginning of line) and '$' (end of line). 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*.[2]

3.8.3. 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,' that is 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 a set of characters 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 a '^', the construct matches all but those characters; thus '[^a–z]' matches anything but a lower-case letter and of course a newline. You must escape any of the characters '^', '[', or '–' in *string* with a preceding '\'.

3.8.4. 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.

[2] 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.

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 between the sequences '\(' and '\)' with side effects in the *substitute* replacement patterns.

3.8.5. 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 '\(' and '\)'.[3] 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.

3.9. Command Reference

The following form is a prototype for all *ex* commands:

address command ! parameters count flags

All parts are optional; the simplest case is the empty command, which displays the next line in the file. To avoid confusion 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*.

(.) 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*.

[3] When nested, parenthesized subexpressions are present, n is determined by counting occurrences of '\(' starting from the left.

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*
edit! *file*

Used to begin an editing session on a new file. Same as **:vi** *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. A 'sensible' file 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.

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 character, it will be supplied and a complaint will be issued. This command leaves the current line '.' at the last line read. If executed from within *open* or *visual*, the current line is initially the first line of the file.

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, for example: '+ /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. In the rare case that the current file is '[Not edited]' this is also noted. You have to use **w!** to write to the file, since *ex* does not want to *write* a file unrelated to the current contents of the buffer.

file *file*

The current filename 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.

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 *Macros* in *Using 'vi', the Visual Display Editor* 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 *Using 'vi' the Visual Display Editor.*

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. 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. 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*. *Ex* also warns you if there are more files in the argument list. Normally, you do want to save your changes, so you should use 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. 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 an accidental hangup of the phone or a system crash or *preserve* command. The system saves a copy of the file you were editing only if you have made changes to the file. 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.

(. , .) 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. If you have modified the current file before giving a *tag* command, you must write it out; giving another *tag* command, specifying no *tag* reuses the previous tag.

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.

vi *file*

Same as **:edit** *file* or **:ex** *file*.

(.) 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 *Using Vi, the Visual Display Editor* for more details. To exit visual mode, type Q.

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.[4] 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* abbr: **x**

If any changes have been made and not written, writes the buffer out. Then, in any case, quits. Same as **wq**, but does not bother to write if there have not been any changes to the file.

[4] 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.

(. , .)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*

Displays a window of text with the specified line at the top. If *type* is '–' the line is placed at the bottom; a '.' places the line 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 terminal, the screen is cleared before display begins unless you give a count which is less than the screen size. The current line is left at the last line displayed. 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.

! *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.

3.10. 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 you then type in lines of text, 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 be aligned with the first non-white character of the previous line. To back the cursor up to the preceding tab stop, type ^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 a '^' and immediately followed by a ^D. This causes the input to be repositioned at the beginning of the line, but retains 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 *rewind*, *stop!* for *stop*, *tag!* for *tag*, *shell* for *!*, and :e # and a :ta!

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 beep or bell.[5] 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*[6]

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 '\'.

[5] Beeping and bell ringing in *open* and *visual* on errors is not suppressed by setting *noeb.*

[6] *Nomagic* for *edit.*

mesg default: mesg

Causes write permission to be turned off to the terminal while you are in visual mode, if *nomesg* is set.

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 ':'.

readonly default: off

If set, writes will unless you use an '!' after the write. Affects **x**, **ZZ**, *autowrite* and anything that writes to guarantee you won't clobber a file by accident. Abbreviate to **ro**.

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**. Can map **q** to **#** and **#1** to something else, and **q1** to something else. If off, can map ^**L** to **1** and ^**R** to ^**L** without having ^**R** map to **1**.

report default: report=5[7]

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.

[7] 2 for *edit*.

scroll default: scroll= 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*).

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 *Using Vi, the Visual Display Editor* 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, similar to the *path* variable of *csh*. Separate the files by spaces, and precede each space with a backslash. Files are searched left to right. Always put *tags* as your first entry. A requested tag is searched for in the specified files, sequentially. By default (even in version 2) 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.

timeout default: on

Causes macros to time out after one second. Turn it off and they wait forever. Use this if you want multi-character macros. If your terminal sends an escape sequence for arrow keys, type ESC twice.

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. Can specify a 12-line window at 300 baud and a 23-line window at 1200 in your EXINIT with: **:set w300=12 w1200=23**. Synonymous with **window** but only at 300, 1200, and 9600 baud.

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. Any number other than 0 is a distance from the right edge of the area where wraps can take place. If you type past the margin, the entire word is rewritten on the next line. Behaves much like fill/nojustify mode in *nroff*. See *Using Vi, the Visual Display Editor* 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.

3.11. 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.

Ex Quick Reference

Entering/Leaving ex

% ex *name*	edit *name*, start at end
% ex +*n* *name*	... at line *n*
% ex −t *tag*	start at *tag*
% ex −r	list saved files
% ex −r *name*	recover file *name*
% ex *name* ...	edit first; rest via :n
% ex −R *name*	read only mode
: x	exit, saving changes
: q!	exit, discarding changes

ex States

Command	Normal and initial state. Input prompted for by :. Your kill character cancels partial command.
Insert	Entered by **a** **i** and **c**. Arbitrary text then terminates with line having only . character on it or abnormally with interrupt.
Open/visual	Entered by open or vi, terminates with Q or ^\.

ex Commands

abbrev	ab	next	n	unabbrev	una
append	a	number	nu	undo	u
args	ar	open	o	unmap	unm
change	c	preserve	pre	version	ve
copy	co	print	p	visual	vi
delete	d	put	pu	write	w
edit	e	quit	q	xit	x
file	f	read	re	yank	ya
global	g	recover	rec	window	z
insert	i	rewind	rew	escape	!
join	j	set	se	shift	<
list	l	shell	sh	print next	CR
map		source	so	resubst	&
mark	ma	stop	st	rshift	>
move	m	substitute	s	scroll	^D

ex Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	*?pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
−	previous	*'x*	marked with *x*
+*n*	*n* forward	''	previous context
%	1,$		

Specifying Terminal Type

% setenv TERM *type*	(for *csh*)
$ TERM=*type*; export TERM	(for *sh*)

See also *tset* in the user's manual.

Some Terminal Types

2621	43	adm31	dw1	h19
2645	733	adm3a	dw2	i100
300s	745	c100	gt40	mime
33	act4	dm1520	gt42	owl
37	act5	dm2500	h1500	t1061
4014	adm3	dm3025	h1510	vt52

Initialising Options

EXINIT	place set's here in environment var.
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Useful Options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and } as typed
slowopen	slow	choke updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer
wrapmargin	wm	automatic line splitting

Scanning Pattern Formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*
[^*str*]	... not in *str*
[*x*-*y*]	... between *x* and *y*
*	any number of preceding

Table of Contents

Chapter 4

Using the ed Line Editor

This chapter[1] describes the editing tools of the *ed* line editor. It provides the newcomer with elementary instructions and exercises for learning the most necessary and common commands and the more advanced user with information about additional editing facilities. The contents include descriptions of appending, changing, deleting, moving, copying and inserting lines of text; reading and writing files; displaying your files; context searching; the global commands; line addressing; and using special characters. There are also brief discussions on the pattern-matching tool *grep*, which is related to *ed*, and on writing scripts.

We assume that you know how to log in to the system and that you have an understanding of what a file is. You must also know what character to type as the end-of-line on your workstation or terminal. This character is the RETURN key in most cases.

Do the exercises in this chapter as you read along. What you enter at the keyboard is shown in **bold face type like this.**

If you need basic information on the Sun system, refer to the *Beginner's Guide to the Sun Workstation*. See *ed* in the *User's Manual for the Sun Workstation* for a nutshell description of the *ed* commands.

4.1. Getting Started

The *ed* text editor is an interactive program for creating and modifying text, using directions that you provide from your workstation. The text can be a document, a program or perhaps data for a program.

We'll assume that you have logged in to your system, and it is displaying the hostname and prompt character, which we show throughout this manual as:

```
logo%
```

To use *ed*, type *ed* at the 'logo%' prompt:

```
logo login:
Password:
Last login: Mon Jul 18 07:50:22 on ttyp0
Sun UNIX 4.2 (Berkeley beta release) (GENERIC) #8: Wed Oct 23 13:45:52 PDT 1983
logo% ed
```

You are now ready to go. *Ed* does not prompt you for information, but waits for you to tell it what to do. First you'll learn how to get some text into a file and later how to change it and make corrections.

---

[1] The material in this chapter is derived from *A Tutorial Introduction to the UNIX Text Editor*, B.W. Kernighan and *Advanced Editing on UNIX*, B.W. Kernighan, Bell Laboratories, Murray Hill, New Jersey.

4.1.1. Creating Text — the Append Command 'a'

Let's assume you are typing the first draft of a memo and starting from scratch. When you first start *ed*, in this case, you are working with a 'blank piece of paper;' there is no text or information present. To supply this text, you either type it in or read it in from a file. To type it in, use the *append* command **a**.

So, to type in lines of text into the buffer, you type an **a** followed by a RETURN, followed by the lines of text you want, like this:

```
logo% ed
a<CR>
Now is the time
for all good men
to come to the aid of their party.
.
```

If you make a mistake, use the DEL key to back up over and correct your mistakes. You cannot go back to a previous line after typing RETURN to correct your errors. The only way to stop appending is to tell *ed* that you have finished by typing a line that contains only a period. It takes practice to remember it, but it has to be there. 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, your file contains the 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, type another **a**, and continue typing.

If you have not used a text editor before, read the following to learn a bit of terminology. If you have used an editor, continue to *Error Messages — '?'*.

In *ed* jargon, the text being worked on is said to be in a work space or 'kept in a buffer.' In effect the buffer is like a piece of paper on which you write things, change some of them, and finally file the whole thing away for another day.

You have learned how to tell *ed* what to do to the text by typing instructions called *commands*. Most commands consist of a single letter, which you must type in lower case, like the append command **a**. Type each command on a separate line. You sometimes precede the command by information about what line or lines of text are to be affected; we discuss this shortly.

As you have seen, *ed* does not respond to most commands; that is, there isn't any prompting or message display like 'ready.' If this bothers you as a beginner, be patient. You'll get used to it.

4.1.2. Error Messages — '?'

When you make an error in the commands you type, *ed* asks you:

```
?
```

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

4.1.3. Writing Text Out as a File — the Write Command 'w'

When you want to save your text for later use, write out the contents of the buffer into a file with the *write* command **w**, followed by the filename you want to write in. The **w** command copies the buffer's contents into the specified file, destroying any previous information on the file. To save the text in a file named *junk*, for example, type:

```
w junk
68
```

Leave a space between the **w** and the filename. *Ed* responds by displaying the number of characters it wrote out, in this case 68. Remember that blanks and the return character at the end of each line are included in the character count. The buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. *Ed* works on a copy of a file at all times, not on the file itself. There is no change in the contents of a file until you type a **w**. Writing out the text into a file from time to time is a good idea to save most of your text should you make some horrible mistake. If you do something disastrous, you only lose the text in the buffer, not the text that was written into the file.

When you want to copy a portion of a file to another name so you can format it separately, use the **w** command. Suppose that in the file being edited you have:

```
.TS
...lots of stuff
.TE
```

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

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

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

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

The point is that **w** can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; give one line number instead of two (we explain line numbers later — see *Specifying Lines in the Editor* for details). For example, if you have just typed a very long, complicated line and you know that you are going to need it or something like it later, then save it — don't re-type it. In the editor, say:

```
a
...lots of stuff...
...very long, complicated line...
.
.w temp
number of characters
a
...more stuff...
.
.r temp
number of characters
a
...more stuff...
.
```

This last example is worth studying to be sure you appreciate what's going on. The **.w temp** writes the very long, complicated line (the current line) you typed to the file called *temp*. The **.r temp** reads that line from *temp* into the file you are editing after the current line 'dot' so you don't have to re-type it.

4.1.4. Leaving 'ed' — the Quit Command 'q'

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

```
w
number of characters
q
logo%
```

The system responds with the hostname prompt. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. Actually, *ed* displays '?' if you try to quit without writing. At that point, write the file if you want; if not, type another **q** to get you out of *ed* regardless.

4.1.5. Exercise: Displaying Your File — the 'cat' Command

Enter *ed*, create some text using **a**, write it out using **w**, and then quit the editor with **q**.

```
a
. . . text . . .
.
w
number of characters
q
logo%
```

Now display the file to see that everything worked. Type the *cat* command with the *junk* filename as the argument in response to the prompt character:

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

Use *cat* when you want to examine a file of less than one screenful. To view a file of more than a screenfull, use the *more* command:

```
logo% more junk
```

To scroll forward in the file, press the space bar. To quit the file, type **q**.

4.1.6. Creating a New File — the Edit Command 'e'

The *edit* command **e** says 'I want to edit a new file called *newfile*, without leaving the editor.' To do this, you type:

```
e newfile
```

The **e** command discards whatever you're currently working on and starts over on *newfile*. It's exactly the same as if you had quit with the **q** command, then re-entered *ed* with a new filename, except that if you have a pattern remembered, a command like '//' will still work. (See *Repeated Searches — '//' and '??'.*)

If you enter *ed* with the command:

```
logo% ed file
```

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

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

and so on does a series of edits on various files without ever leaving *ed* and without typing the name of any file more than once. As an aside, if you examine the sequence of commands here, you can see why you can use **e** as a synonym for *ed*.

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 **w** in a previous session. The *edit* command **e** also fetches the entire contents of a file into the buffer. So if you had saved the three lines 'Now is the time', etc., with **w** in an earlier session, the *ed* command **e** fetches the entire contents of the file *junk* into the buffer, and responds with the number of characters in *junk*:

```
logo% e junk
68
```

If anything was already in the buffer, it is deleted first.

If you use **e** to read a file into the buffer, you do not need to use a filename after a subsequent **w** command; *ed* remembers the last filename used in an **e** command, and **w** will write on this file. Thus a good way to operate is:

```
logo% ed
e file
number of characters
[editing session]
w
number of characters
q
logo%
```

This way, you can simply say **w** from time to time, and be secure that you are writing into the proper file each time.

4.1.7. Exercise: Trying the 'e' Command

Experiment with the **e** command — try reading and displaying 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 filename wrong, or perhaps because you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that:

```
logo% ed filename
```

is equivalent to:

```
logo% ed
e filename
number of characters
```

4.1.8. Checking the Filename — the Filename Command 'f'

You can find out the remembered filename at any time with the **f** command; just type **f** without a filename. You can also change the name of the remembered filename with **f**; this following sequence guarantees that a careless **w** command will write on *junk* instead of *precious*. Try:

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

You can find out at any time what filename *ed* is remembering by typing the *file* command **f**. In this example, if you type **f**, *ed* replies:

```
logo% ed junk
68
f
junk
```

4.1.9. 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. To do this, use the *read* command **r**. The command:

```
r junk
68
```

reads the file *junk* into the buffer, adding it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
e junk
68
r junk
68
w
136
q
logo%
```

the buffer contains *two* copies of the text or six lines in this case. Like **w** and **e**, **r** displays the number of characters read in after the reading operation is complete. Now check the buffer contents with *cat*:

```
logo% cat junk
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.
logo%
```

Generally speaking, you won't use **r** as much as **e**.

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

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

The data contained in *table* has to go there so *nroff* or *troff* will format it properly. Now what? This one is easy. Edit *memo*, find 'Table 1', and add the file *table* right there:

```
logo% ed memo
/Table 1/
Table 1 shows that ... (response from ed)
.r table
```

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

4.1.10. Printing the Buffer Contents — the Print Command 'p'

To print or 'display' the contents of the buffer or parts of it on the screen, use the *print* command **p**. To do this, specify the lines where you want the display to begin and where you want it to end, separated by a comma, and followed by **p**. Thus to show the first two lines of the buffer, for example, say:

```
1,2p   (starting line=1, ending line=2 p)
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use **1,3p** if you knew there were exactly three lines in the buffer. But in general, you don't know how many lines 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 to display *all* the lines in the buffer, line 1 to last line:

```
1,$p
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.
```

If you want to stop the display of more than one screenfull before it is finished, type the INTERRUPT character ^C (or the DEL key).

```
^C
?
```

Ed waits for the next command.

To display the *last* line of the buffer, you can use:

```
$,$p
to come to the aid of their party.
```

or abbreviate it to:

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

You can show any single line by typing the line number followed by a **p**. So, to display the first line of the buffer, type:

```
1p
Now is the time
```

In fact, *ed* lets you abbreviate even further: you can display any single line by typing *just* the line number — there is no need to type the letter **p**. So if you say:

```
2
for all good men
```

ed displays the second line of the buffer.

You can also use '**$**' in combinations to display the last two lines of the buffer, for example:

```
$-1,$p
for all good men
to come to the aid of their party.
```

This helps when you want to see how far you got in typing.

4.1.11. Exercise: Trying the 'p' Command

As before, create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't show line 0 or a line beyond the end of the buffer, and that attempts to show a buffer in reverse order don't work. For example, you get an error message if you type:

```
3,1p
?
```

4.1.12. Displaying Text — the List Command 'l'

Ed provides two commands for displaying the contents of the lines you're editing. You are familiar with the **p** command that displays lines of text. Less familiar is the *list* command **l** (the letter '*l*'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will show each tab as > and each backspace as <. A sample display of a random file with tab characters and backspaces is:

```
l
Now is the >> time for << all good men
```

This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing. Any line that exceeds 72 characters is displayed on multiple lines. Each printed line except the last is terminated by a backslash ' \', so you can tell it was folded. This is useful for displaying long lines on small terminal screens. A sample output of a folded line is:

```
l
This is an example of using the 'l' command to display a very long line that \
has more than 72 characters ...
```

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

4.1.13. The Current Line — 'Dot' or '.'

Suppose your buffer still contains the six lines as above, and that you have just typed:

```
1,3p
Now is the time
for all good men
to come to the aid of their party.
```

Ed has displayed the three lines for you. Try typing just a p to display:

```
p       (no line numbers)
to come to the aid of their party.
```

which is the third line of the buffer. In fact it is the last or most recent line that you have done anything with. (You just displayed it!) You can repeat p without line numbers, and it will continue to display 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 displayed) so that you can use it instead of an explicit line number. You refer to this most recent line by the shorthand symbol:

```
.  (pronounced 'dot')
to come to the aid of their party.
```

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 display all the lines from and including the current line to the end of the buffer.

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

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 displayed; that is, after this command sets both '.' and '$' refer to the last line of the file, line 6.

Dot is most useful in combinations like:

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

This means 'show the next line' and is a handy way to step slowly through a buffer. You can also say:

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

which means 'show the line *before* the current line.' Use this to go backwards if you wish. Another useful one is something like:

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

which shows 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:

```
.=
3
```

Let's summarize some things about **p** and dot. Essentially you can precede **p** by 0, 1, or 2 line numbers. If you do not give a line number, **p** shows the 'current line,' the line that dot refers to. If there is one line number given with or without the letter **p**, it shows that line and dot is set there; and if there are two line numbers, it shows all the lines in that range, and sets dot to the last line displayed. If you specify two line numbers, the first can't be bigger than the second.

Typing a single RETURN displays the next line — it's equivalent to **.+1p**. Try it. Try typing a **–**; you will find that it's equivalent to **.–1p**.

4.1.14. Deleting Lines — the Delete Command 'd'

Suppose you want to get rid of the three extra lines in the buffer. To do this, use the *delete* command **d**. The **d** command is similar to **p**, except that **d** deletes lines instead of displaying them, You specify the lines to be deleted for **d** exactly as you do 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
Now is the time
for all good men
to come to the aid of their party.
```

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

4.1.15. Exercise: Experimenting

Experiment with **a**, **e**, **r**, **w**, **p** and **d** until you are sure you know what they do, and until you understand how to use dot, '$' and the line numbers.

If you are adventurous, try using line numbers with **a**, **r** and **w** as well. You will find that **a** appends lines *after* the line number that you specify rather than after dot; that **r** reads a file in *after* the line number you specify and not necessarily at the end of the buffer; and that **w** writes out exactly the lines you specify, not necessarily the whole buffer. These variations are handy, for instance, for inserting a file at the beginning of a buffer:

```
0r filename
number of characters
```

Ed indicates the number of characters read in. You can enter lines at the beginning of the buffer by saying:

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

Or you can write out the lines you specify with **w**. Notice that **.w** is *very* different from:

```
.
w
number of characters
```

4.1.16. Modifying Text — the Substitute Command 's'

One of the most important commands is the *substitute* command **s**. Use **s** to change individual words or letters within a line or group of lines. For example, you can correct 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'. *Ed* does not display the result automatically, so verify that it works with:

```
p
Now is the time
```

You get what you wanted. Notice that dot has been set to the line where the substitution took place, since **p** printed that line. The **s** command always sets dot in this way.

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, read on below. 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!)

You can precede any **s** command by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus, to change the *first* occurrence of 'mispell' to 'misspell' on every line of the file, type:

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

But to change *every* occurrence in every line, type:

1,$s/mispell/misspell/g

This is more likely what you wanted in this particular case.

Note: Be careful that this is exactly what you want to do. Unless you specify the substitution specifically, globally changing the string 'the', will also change every instance of those characters, including '*other*', etc.

If you do not give any line numbers, **s** assumes you mean 'make the substitution on line dot,' so it changes things only on the current line. You will see that a very common sequence is to correct a mistake on the current line, and then display the line to make sure everything is all right:

s/something/something else/p
line with something else

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.

You can also say:

s/ . . . //

which means 'change the first string of characters to *nothing*;' that is, remove the first string of characters. Use this sequence for deleting extra words in a line or removing extra letters from words. For instance, if you had:

Nowxx is the time

To correct this, say:

s/xx//p
Now is the time

Notice that '//' (two adjacent slashes) means 'no characters,' not a blank. There *is* a difference! (See *Repeated Searches* for another meaning of '//'.)

If you want to replace the *first* 'this' on a line with 'that', for example, use:

s/this/that/

If there is more than one 'this' on the line, a second form with the trailing *global* command **g** changes *all* of them:

s/this/that/g

The general format is:

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*.

You can follow either form of the **s** command by **p** or **l** to display or list the contents of the line.

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

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

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

4.1.17. The Ampersand '&'

The '&' is a 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.' Use it 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.'

You can use the '&' several times in a line:

```
s/.*/&? &!!/
Now is the time? Now is the time!!
```

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

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

makes:

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

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

```
the end of the world
```

you can type:

```
/world/s//& is at hand/
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.

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

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/\&/
```

converts the word 'ampersand' into the literal symbol '&' in the current line. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to put parentheses around a line, regardless of its length, use:

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

4.1.18. Exercise: Trying the 's' and 'g' Commands

Experiment with **s** and **g**. 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
on the other side of the coin
```

4.1.19. Undoing a Command — the Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. Use the *undo* command **u** to undo the last substitution. This restores the last line that was substituted to its previous state. For example, study the following example:

```
s/party/country/
p
to come to the aid of their country.
u
p
to come to the aid of their party.
```

4.2. Changing and Inserting Text — the 'c' and 'i' Commands

This section discusses the *change* command **c**, which changes or replaces a group of one or more lines, and the *insert* command **i**, which inserts a group of one or more lines.

The **c** command replaces a number of lines with different lines, which you type in at the workstation. 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 '.' 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 you only specify one line in the **c** command, 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 . . .
.

inserts the given text *before* the next line that contains 'string', that is, 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.

4.2.1. Exercise: Trying the 'c' Command

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 to append *after* the given line, you type:

line-number **a**
. . . text . . .
.

while to insert *before* it, you type:

line-number
. . . text . . .

Observe that if you do not give a line number, **i** inserts before line dot, while **a** appends after line dot.

4.3. Specifying Lines in the Editor

To specify what lines are to be affected by the editing commands, you use *line addressing*. There are several methods, which are described below.

4.3.1. Context Searching

One way is *context searching*. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

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.
```

If you want to find the line that contains 'their' so you can change it to 'the'. 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 contains 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.

For example, to locate the next occurrence of the characters between slashes ('their'), type:

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

To search for a line that contains a particular string of characters, the general format is:

/string of characters we want to find/

This is sufficient to find the desired line. It also sets dot to that line and displays the line for verification. '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* displays the error message:

```
?
```

Otherwise it shows the line it found.

Less familiar is the use of:

?thing?

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

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command. You can do both the search for the desired line *and* a substitution all at once, like this:

```
/their/s/their/the/p
to come to the aid of the party.
```

There were three parts to that last command: a context search for the desired line, the substitution, and displaying 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 you can use them by themselves to find and show a desired line, or as line numbers for some other command, like **s**. We use them both ways in the examples above.

4.3.2. Exercise: Trying Context Searching

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. You can also use context searching with with **r**, **w**, and **a**.

If you get funny results with any of the characters:

^ . $ [* \ &

read the section on *Special Characters*.

4.3.3. Specifying Lines with Address Arithmetic — '+' and '–'

Another area in which you can save typing in specifying lines is to use '–' and '+' as line numbers by themselves. To move back up one line in the file, type:

```
–
```

In fact, you can string several minus signs together to move back up that many lines:

```
---
```

moves up three lines, as does '–3'. Thus:

```
–3,+3p
```

is also identical to the examples above.

Since '–' is shorter than '.–1', use it to change 'bad' to 'good' on the previous line and on the current line.

```
–,.s/bad/good/
```

You can use '+' and '–' in combination with searches using '/.../' and '?...?', and with '$'. To find the line containing 'thing', and position you two lines before it, type:

```
/thing/--
```

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

```
$-1
```

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

```
$-5,$p
```

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

Convenience dictates the choice. You could display 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 you can use it wherever a line number is needed.

As another example:

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

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

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

is identical in meaning.

4.3.4. Repeated Searches — '//' and '??'

Suppose you ask for the search:

```
/horrible thing/
```

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

```
//
```

as a shorthand for 'the previous thing that was searched for', whatever it was. You can repeat this as many times as necessary. You can also go backwards by typing:

```
??
```

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

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

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

To go backwards and change a line, say:

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

You can also use it as the first string of the substitute command, as in:

```
/string1/s//string2/
```

which finds the next occurrence of 'string1' and replaces it by 'string2'. This can save a lot of typing.

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

```
//s//& &/p
```

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

4.3.5. Default Line Numbers and the Value of Dot

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

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

```
/thing/
```

you are left pointing at the next line that contains 'thing'. No address is required with commands like **s** to make a substitution on that line. Addresses are also not required with **p** to show it, **l** to list it, **d** to delete it, **a** to append text after it, **c** to change it, or **i** to insert text before it.

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

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

The line-changing commands **a**, **c** and **i** by default all affect the current line. If you do not give a line number with them, the **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

The **a**, **c**, and **i** commands behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for

typing and editing on the fly. For example, you can say:

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

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

```
a
... text ...
... horrible botch ...     (major error)

c                          (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you do not add *any* lines with **a**, **c** or **i**.

The **r** command reads a file into the text being edited, either at the end if you do not give an address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **0r** to read a file in at the beginning of the text. You can also say **0a** or **1i** to start adding text at the beginning.

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

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

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

With the **s** command, the rule is simple; you are left positioning on the last line that got changed. If there were no changes, dot doesn't move.

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

```
x1
x2
x3
```

Then to display the third line, which is the last one changed, type:

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

But if the three lines had been:

```
x1
y2
y3
```

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

4.3.6. Combining Commands — the Semicolon ';'

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

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

Starting at line 1, one would expect that the command:

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

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

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

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

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

Use the semicolon when you want to find the *second* occurrence of something. For example, to find the second occurrence of 'thing', you can say:

```
/thing/
line with 'thing'
//
second line with 'thing'
```

But this displays the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to find the first occurrence of 'thing', set dot to that line, then find the second and display only that:

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

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

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

We leave you to try showing the third or fourth or ... in either direction.

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

1;/thing/

This search fails if 'thing' occurs on line 1. But it is possible to say:

0;/thing/

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

4.3.7. Interrupting the Editor

As a final note on what dot gets set to, be aware that if you type an INTERRUPT (^C is the default, but your terminal may be set up with the DELETE, RUBOUT or BREAK keys) while *ed* is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in the middle of execution in some clean but unpredictable state; hence it is not usually wise to stop them. Dot may or may not be changed.

Displaying is more clear cut. Dot is not changed until the display is done. Thus if you display lines until you see an interesting one, then type ^C, you are *not* sitting on that line or even near it. Dot is left where it was when the **p** command was started.

4.4. Editing All Lines — the Global Commands 'g' and 'v'

Use the *global* command **g** to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example, to display all lines that contain 'peling', type:

g/peling/p

As another example:

g/^\./p

displays all the formatting commands in a file lines that begin with '.'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; the same rules and limitations apply.

As a more useful command which makes the substitution everywhere on the line, then displays each corrected line, type:

g/peling/s//pelling/gp

Compare this to the command line, which only shows the last line substituted:

1,$s/peling/pelling/gp

Another subtle difference is that the **g** command does not give a '?' if 'peling' is not found where the **s** command will.

The substitute command is probably the most useful command that can follow a global because you can use this to make a change and display each affected line for verification. For example, you can change the word 'Sun' to 'SUN' everywhere in a file, and verify that it really worked, with:

```
g/Sun/s//SUN/gp
```

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

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

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

deletes all lines that begin with '.', and:

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

deletes all empty lines.

The **v** command is identical to **g**, except that it operates on those line that do *not* contain an occurrence of the pattern; that is, **v** 'inverts' the process, so:

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

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

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

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

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

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

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

displays all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

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

4.4.1. Multi-line Global Commands

You can use several commands at once, including **a**, **c**, **i**, **r**, **w**, but not **g**; in this case, every line except the last must end with a backslash ' \ '. For example, to make changes in the lines before and after each line that contains 'xxx', and then display all three lines, say:

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

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

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

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

Watch out for the command:

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

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

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

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

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

You do not need a final line containing a '.' to terminate the **i** command, unless you are using further commands under the global command. On the other hand, it does no harm to put it in either.

4.5. Special Characters

Certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. You may have noticed that things just don't work right when you use some characters like '.', '*', '$', and others in context searches and with the substitute command. These special characters are called *metacharacters*. 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 is:

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

4.5.1. Matching Anything — the Dot '.'

Use the 'dot' metacharacter '.' to match any single character. For example, to find any line where 'x' and 'y' occur separated by a single character, type:

```
/x.y/
```

You may get any of:

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

and so on.

On the left side of a substitute command, or in a search with '/.../', '.' stands for *any* single character.

Since '.' matches a single character, it gives you a way to deal with funny characters that l displays. Suppose you have a line that, when displayed with the l command, appears as:

```
.... th07is ....
```

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

The most obvious solution is to try:

```
s/07//
```

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

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

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

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

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

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

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

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

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

```
Now is the time.
```

the result will be:

```
.s/././
.ow is the time.
```

which is probably not what you intended.

4.5.2. Specifying Any Character — the Backslash ' \'

The backslash character ' \' is special to *ed* as noted in the description of the ampersand. 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/
```

changes '\.*' into 'backslash dot star'.

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

Now is the time.

into:

Now is the time?

Use the backslash ' \' here as well to turn off any special meaning that the next character might have; in particular, ' \.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in 'Now is the time.', type:

```
s/\./?/p
Now is the time?
```

Ed treats the pair of characters '\.' as a single real period.

You can also use the backslash when searching for lines that contain a special character. Suppose you are looking for a line that contains:

```
.PP
```

The search for '.PP' finds:

```
/.PP/
THE APPLICATION OF ...
```

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

```
/\.PP/
```

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

You can also use the backslash to turn off special meanings for characters other than ' \' and '.'. For example, consider finding a line that contains a backslash. The search:

```
/\/
```

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

```
/\\/
```

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

```
/\//
```

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

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

```
\x\.\y
```

into the line:

```
\x\y
```

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

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

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

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

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

```
s:/::g
```

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

4.5.3. Specifying the End of Line — the Dollar Sign '$'

The dollar-sign '$' means the end of a line:

```
/string$/
```

only finds an occurrence of 'string' that is at the end of some line. This implies, of course, that:

```
/^string$/
```

finds a line that contains just 'string', and:

```
/^.$/
```

finds a line containing exactly one character.

As an obvious use, suppose you have the line:

```
Now is the
```

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

```
s/$/ time/p
Now is the time
```

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

```
Now is thetime
```

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

```
s/,$/./p
Now is the time, for all good men,
```

The '$' sign here provides context to make specific which comma you mean. Without it, of course, **s** operates on the first comma to produce:

```
s/,/./p
Now is the time. for all good men,
```

As another example, to convert:

Now is the time.

into:

Now is the time!

as you did earlier, you can use:

```
s/.$/?/p
Now is the time?
```

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

```
$s/$/$/
```

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

4.5.4. Specifying the Beginning of the Line — the Circumflex '^'

The circumflex '^' signifies the beginning of a line. Thus:

```
/^string/
string
```

finds 'string' only if it is at the beginning of a line, but not:

the string...

You will in all likelihood find several lines that contain the desired string before arriving at the one you want, unless you specify the string more exactly. You narrow the context, and thus arrive at the desired one more easily if you type something like:

```
/^the/
the
```

to find 'the' at the beginning of the line.

You can also use '^' to insert something at the beginning of a line. For example, to place a space at the beginning of the current line, type:

```
s/^/ /
```

You can combine metacharacters. To search for a line that contains *only* the characters '.PP' by typing:

```
/^\.PP$/
.
```

4.5.5. Matching Anything — the Star '*'

Suppose you have a line that looks like this:

text x y *text*

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

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

s/x *y/x y/

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

You can use the star with any character, not just the space. If the original example was instead:

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

then you can replace all '–' signs by a single space with the command:

s/x-*y/x y/

Finally, suppose that the line was:

text x.....y *text*

Can you see what trap lies in wait for the unwary? What will happen if you blindly type:

s/x.*y/x y/

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

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

then saying:

s/x.*y/x y/

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

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

s/x\.*y/x y/

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

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. Use this like:

s/.*/stuff/

which changes an entire line, or:

```
s/,/./p
Now is the time. for all good men,
```

As another example, to convert:

```
Now is the time.
```

into:

```
Now is the time?
```

as you did earlier, you can use:

```
s/.$/?/p
Now is the time?
```

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

```
$s/$/$/
```

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

4.5.4. Specifying the Beginning of the Line — the Circumflex '^'

The circumflex '^' signifies the beginning of a line. Thus:

```
/^string/
string
```

finds 'string' only if it is at the beginning of a line, but not:

```
the string...
```

You will in all likelihood find several lines that contain the desired string before arriving at the one you want, unless you specify the string more exactly. You narrow the context, and thus arrive at the desired one more easily if you type something like:

```
/^the/
the
```

to find 'the' at the beginning of the line.

You can also use '^' to insert something at the beginning of a line. For example, to place a space at the beginning of the current line, type:

```
s/^/ /
```

You can combine metacharacters. To search for a line that contains *only* the characters '.PP' by typing:

```
/^\.PP$/
.
```

4.5.5. Matching Anything — the Star '*'

Suppose you have a line that looks like this:

text x y *text*

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

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

s/x *y/x y/

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

You can use the star with any character, not just the space. If the original example was instead:

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

then you can replace all '–' signs by a single space with the command:

s/x–*y/x y/

Finally, suppose that the line was:

text x.....y *text*

Can you see what trap lies in wait for the unwary? What will happen if you blindly type:

s/x.*y/x y/

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

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

then saying:

s/x.*y/x y/

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

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

s/x\.*y/x y/

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

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. Use this like:

s/.*/stuff/

which changes an entire line, or:

```
s/,/./p
Now is the time. for all good men,
```

As another example, to convert:

Now is the time.

into:

Now is the time!

as you did earlier, you can use:

```
s/.$/!/p
Now is the time!
```

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

```
$s/$/$/
```

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

4.5.4. Specifying the Beginning of the Line — the Circumflex '^'

The circumflex '^' signifies the beginning of a line. Thus:

```
/^string/
string
```

finds 'string' only if it is at the beginning of a line, but not:

the string...

You will in all likelihood find several lines that contain the desired string before arriving at the one you want, unless you specify the string more exactly. You narrow the context, and thus arrive at the desired one more easily if you type something like:

```
/^the/
the
```

to find 'the' at the beginning of the line.

You can also use '^' to insert something at the beginning of a line. For example, to place a space at the beginning of the current line, type:

```
s/^/ /
```

You can combine metacharacters. To search for a line that contains *only* the characters '.PP' by typing:

```
/^\.PP$/
.
```

4.5.5. Matching Anything — the Star '*'

Suppose you have a line that looks like this:

text x y *text*

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

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

```
s/x *y/x y/
```

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

You can use the star with any character, not just the space. If the original example was instead:

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

then you can replace all '–' signs by a single space with the command:

```
s/x-*y/x y/
```

Finally, suppose that the line was:

text x.....y *text*

Can you see what trap lies in wait for the unwary? What will happen if you blindly type:

```
s/x.*y/x y/
```

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

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

then saying:

```
s/x.*y/x y/
```

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

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

```
s/x\.*y/x y/
```

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

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. Use this like:

```
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.

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

```
Now is the time for all good men ....
s/ for.*/./p
Now is the time.
```

The '.*' eats up everything after the 'for'.

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

text xy *text* x y *text*

and you said:

s/x *y/x y/

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

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

/x *y/

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The following command does not produce what was intended:

```
abcdef
s/x*/y/g
p
yaybycydyeyfy
```

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

s/xx*/y/g

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

4.5.6. Character Classes — Brackets '[ ]'

The '[' and ']' brackets form 'character classes'; for example, to match any single digit, use:

/[0123456789]/

Any one of the characters inside the braces will cause a match. You can abbreviate this to '[0–9]'. Or for example, to match zero or more digits (an entire number), to delete all digits

from the beginning of all lines, type:

```
1,$s/^[0123456789]*//
```

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

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

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

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

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

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

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0–9]; similarly, [a–z] stands for the lower-case letters, and [A–Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. To do this, begin the class with a '^' to stand for 'any character *except* a digit':

```
[^0-9]
```

Thus you might find the first line that doesn't begin with a tab or space by a search like:

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

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that to find a line that doesn't begin with a circumflex, you type:

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

4.6. Cutting and Pasting with the Editor

Ed has commands for manipulating individual lines or groups of lines in files.

4.6.1. Moving Lines Around

There are several ways to move text around in a file.

4.6.2. Moving Text Around — the Move Command 'm'

Use the *move* command m for cutting and pasting — you can 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
```

This is the brute force way; that is, you write the paragraph onto a temporary file, read in the temporary file at the end, and then delete it from its current position. As another example, consider:

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

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

But you can do it a lot easier with **m**, so you can do a whole operation at one crack.

```
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.

If you try:

```
1,5m3
?
```

ed reminds you that you can't do this.

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

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

line1, line2 **m** *line3*

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

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

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

That's all.

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

```
m+
```

If you are positioned on the second line, and want to do the reverse, type:

```
m--
```

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

4.6.3. Substituting Newlines

You can split a single line into two or more shorter lines by 'substituting in a newline.' As the simplest example, suppose a line has gotten unmanageably long because of editing or merely because it was unwisely typed. If it looks like:

text xy *text*

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

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

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

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

text a very big *text*

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

```
s/ very /\
.ul\
very\
/
```

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

4.6.4. Joining Lines — the Join Command 'j'

You may also join lines together, but use the *join* command **j** for this instead of **s**. Given the lines:

Now is
the time

and supposing that dot is set to the first of them, then the command:

j

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a **j** command joins line dot to line dot+ 1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example:

1,$jp

joins all the lines into one big one and displays it.

4.6.5. Rearranging a Line with '\(... \)'

Skip this section if this is the first time you're reading this chapter. Recall that '&' stands for whatever was matched by the left side of an **s** command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

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

Smith, A. B.
Jones, C.

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

A. B. Smith
C. Jones

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

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

The command:

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

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

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

4.6.6. Marking a Line — the Mark Command 'k'

You can mark a line with a particular name so you can refer to it later by name, regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**. To mark the current line with the name 'x', use:

```
kx
```

If a line number precedes the **k**, that line is marked. The mark name must be a single lowercase letter. Now you can refer to the marked line with the address:

```
'x
```

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

```
'a,'bm.
```

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

4.6.7. Copying Lines — the Transfer Command 't'

We mentioned earlier the idea of saving a line that was hard to type or used often, to cut down on typing time. Of course this can be more than one line, in which case the saving is presumably even greater.

Ed provides another command, called **t** (*transfer*) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to **m**, except that instead of moving lines, it simply duplicates them at the place you named. Thus, to duplicate the entire contents that you are editing, use:

```
1,$t$
```

A more common use for **t** is for creating a series of lines that differ only slightly. For example, you can say:

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

and so on.

4.7. Escaping to the Shell with '!'

Sometimes it is convenient to be able to temporarily escape from the editor to use some Shell command without leaving the editor. Use the '!' (escape) command to do this.

To suspend your current editing state and execute the Shell command you asked for, type:

```
!any Shell command
!
```

When the command finishes, *ed* will signal you by displaying another '!'; at that point, you can resume editing.

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

4.8. Supporting Tools

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how *ed* works, because they are all based on the editor. This section gives some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. For more information on each, refer to the *User's Manual for the Sun Workstation*.

4.8.1. Editing Scripts

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

For example, suppose you want to change every 'Sun' to 'SUN' and every 'System' to 'SYSTEM' in a large number of files. Then put into a file, which we'll call *changes*, the lines:

```
g/Sun/s//SUN/g
g/System/s//SYSTEM/g
w
q
```

Now you can say:

```
logo% ed file1 <script
logo% ed file2 <script
...
```

This causes *ed* to take its commands from the prepared script called *changes*. Notice that you have to plan the whole job in advance.

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

4.8.2. Matching Patterns with 'grep'

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

The program *grep* gets around these limitations. The search patterns that are described in this chapter are often called 'regular expressions', and 'grep' stands for 'general regular expression, print.' That describes exactly what *grep* does — it displays every line in a set of files that

contains a particular pattern. Thus, to find 'thing' wherever it occurs in any of the files *file1*, *file2*, etc., type:

```
logo% grep 'thing' file1 file2 file3 ...
                . . .
logo%
```

Grep also indicates the file in which the line was found, so you can later edit it if you like.

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

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

```
logo% grep -v 'thing' file1 file2 ...
                . . .
logo%
```

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

```
logo% grep x file... | grep -v y
                . . .
logo%
```

The notation '|' is a 'pipe', which causes the output of the first command to be used as input to the second command; see the *Beginner's Guide to the Sun Workstation* for an introduction to 'piping.' See the *User's Manual for the Sun Workstation* for details on *grep*.

4.9. 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 filename. 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. Type a '.' on a new line to terminate appending. Dot is set to the last line appended.

c Change the specified lines to the new text which follows. Type a '.' as with **a** to terminate the change. If no lines are specified, replace line dot. Dot is set to last line changed.

d Delete the lines specified. If none is 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 use 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*

executes 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 Display specified lines. If none is specified, display line dot. A single line number is equivalent to *line-number* **p**. Type a single RETURN to show '.+ 1', the next line.

q Quit *ed*. This 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 the end unless specified elsewhere. Dot is set to last line read.

s The command:

s/string1/string2/

substitutes the characters 'string2' into 'string1' 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. An **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.

.= Show value of dot. An '= ""' by itself shows the value of '$.'.

! The line:

!*command-line*

executes *command-line* as a Sun UNIX command.

/-----/ Context search. Search for next line which contains this string of characters and display 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 '$.'.

Table of Contents

Chapter 5

Using sed, the Stream Text Editor

This chapter[1] describes *sed*, the non-interactive context or *stream* editor. Use *sed* for editing files too large for comfortable interactive editing, editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode, and performing multiple global editing functions efficiently in one pass through the input. Because the default mode is to apply edit commands globally, and because its output is to the Standard Output, your workstation or terminal screen, *sed* is good for making changes of a transient nature, rather than permanent modifications to a file.

You can create a complicated editing script separately and use it as a command file. For complex edits, this saves considerable typing, and its attendant errors. Running *sed* from a command file is much more efficient than any interactive editor even if that editor can be driven by a pre-written script.

Whereas the *ed* editor copies your original file into a buffer *sed* does not use temporary files, so you can edit any size file. The only space requirement is that the input and output fit simultaneously into the available second storage. Additionally, *ed* lets you explore the text in whatever order you want, while *sed* works on your file from beginning to end, and allows you no choice of edit commands once you have started it. Basically *sed* passes some data through a set of transformations called editor *functions*.

By default *sed* copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. You can modify this behavior by adding a command-line option; see *Command Options* below.

As a lineal descendant of the *ed* editor, *sed* recognizes basically the same regular expressions as *ed*. The range of pattern matches is called the *pattern space*. Ordinarily, the pattern space is one line of text, but you can read more than one line into the pattern space if necessary. But because of the differences between interactive and non-interactive operation, *ed* and *sed* are different enough that even experienced *ed* users should read this chapter. You cannot use relative addressing with *sed* as you can with an interactive editor because *sed* operates a line at a time. *Sed* also does not give you any immediate verification that a command has done what was intended.

Refer to *Using the 'ed' Line Editor* for more information on *ed* and to the descriptions of *sed* and *ed* in the *User's Manual for the Sun Workstation*.

5.1. Using sed

The general format of an editing command is:

logo% **sed** [*line1*[,*line2*]] *function* [*arguments*]

[1] The material in this chapter is derived from *Sed — a Non-Interactive Text Editor*, L.E. McMahon, Bell Laboratories, Murray Hill, New Jersey.

There is an optional line address, or two line addresses separated by a comma, a single-letter edit function, followed by other arguments, which may be required or optional, depending on which function you use. See *Specifying Lines for Editing* for the format of line addresses. Any number of blanks or tabs may separate the line addresses from the function. *Sed* ignores tab characters and spaces at the beginning of lines. The function must be present; the available commands are discussed in *Functions* under each individual function name. You can either put the edit commands on the *sed* command line or put the commands in a file, which is then applied to the file you want to edit. If the commands are few and simple, put them on the *sed* command line. For example, assume the following input text in a file called *kubla*:

```
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.
```

Let's copy the first two lines of input as a simple example:

```
logo% sed 2q kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

As another example, suppose that you want to change the 'Khan' to 'KHAN.' Then the command:

```
logo% sed s/Khan/KHAN/g kubla
```

applies the command 's/Khan/KAN/' to all lines from *kubla* and copies all lines to the standard output. The advantage of using *sed* in such a case is that you can use it with input too large for *ed* to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

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

```
logo% sed -f cmdfile input-files...
```

5.1.1. Command Options

Sed has three options that modify *sed*'s action. If you invoke *sed* with the **-f** (file) option, the edit commands are taken from a file. For example:

```
logo% sed -f edcomds oldfile > newfile
logo%
```

The name of the file containing the edit commands must immediately follow the **-f** option. Here, the edit commands in the *edcomds* file are applied to the file *oldfile*, and the standard output is redirected to *newfile*.

You use the **-e** (edit) option to place editing commands directly on the *sed* command line. If you are only using one edit command, you can omit the **-e**, but we include it in the example below for instructive purposes. For example, to delete a line containing the string 'Khan' from *kubla*, you type:

```
logo% sed -e /Khan/d kubla > newkubla
logo%
```

If you put more than one edit command on the *sed* command line, each one must be preceded by **-e**. For example:

```
logo% sed -e /Khan/d -e s/decree/DECREE/ newkubla
logo%
```

You can also use both the **-e** and the **-f** options at the same time.

Sed normally copies all input lines that are changed by the edit operation to the output. If you want to suppress this normal output, and have only specific lines appear on the output, use the **-n** option with the **p** (print) flag. For example:

```
logo% sed -n -e s/to/by/p kubla
Through caverns measureless by man
Down by a sunless sea.
logo%
```

As a quick reference, these options are:

-f Use the next argument as a filename; the file should contain one editing command to a line.

-e Use the next argument as an editing command.

-n Send only those line specified by **p** functions or **p** functions after substitute functions (see *Input-Output Functions*) to the output.

5.2. Editing Commands Application Order

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a moderately efficient form for execution 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.

You can change the default linear order of application of editing commands by the flow-of-control commands, **t** and **b** (see *Flow-of-Control Functions*). Even when you change the order of application by these commands, it is still true that the input line to any command is the output of any previously applied command.

5.3. Specifying Lines for Editing

Use addresses to select lines in the input file(s) to apply the editing commands to. Addresses may be either line numbers or context addresses.

Group one address or address-pair with curly braces '{ }' to control the application of a group of commands. See *Flow-of-Control Functions* for more on this.

5.3.1. 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 or '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.

5.3.2. Context Addresses

A context address is a pattern or *regular expression* enclosed in slashes ('/'). *Sed* recognizes the regular expressions that are constructed as follows:

ordinary character
: An ordinary character (not one of those discussed below) is a regular expression, and matches that character.

^
: A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.

$
: A dollar-sign '$' at the end of a regular expression matches the null character at the end of a line.

\n
: The characters '\n' match an embedded newline character, but not the newline at the end of the pattern space.

.
: A period '.' matches any character except the terminal newline of the pattern space.

*
: A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.

[*character string*]
: 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 a circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.

concatenation
: A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

\(\)
: A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described in *The Substitute Function 's'* and immediately below.

\d
: This stands for 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.

null
: The null regular expression standing alone (such as, '//') is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal, that is, 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.

5.3.3. Number of Addresses

The commands described in *Functions* can have 0, 1, or 2 addresses. Specifying more than the maximum number of addresses allowed is 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 that match that address. If a command has two addresses, it is applied to the inclusive range defined by those two addresses.

The command is applied to the first line that 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.

A comma separates two addresses.

For example:

/an/ matches lines 1, 3, 4 in our sample *kubla* file

In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man

/an.*an/ matches line 1 In Xanadu did Kubla Khan

/^an/ matches no lines

/./ matches all lines 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.

/\./ matches line 5 Down to a sunless sea.

/r*an/ matches lines 1,3, 4 (number = zero!) In Xanadu did Kubla Khan Where Alph, the sacred river, ran Through caverns measureless to man

/\(an\).*\1/ matches line 1 In Xanadu did Kubla Khan

5.4. Functions

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is enclosed in parentheses, followed by the single character function name and possible arguments in italics. The summary provides an expanded English translation of the single-character name, and a description of what each function does.

5.4.1. Whole Line Oriented Functions

The functions that operate on a whole line of input text are as follows:

*(2)***d** Delete lines. The **d** function deletes from the file all those lines matched by its address(es); that is, it does not write the indicated lines to the output, No further commands are attempted on 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 writes the argument *text* to the output after the line matched by its address. The **a** function 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, 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 change 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**, put a newline hidden by a backslash after **c**; interior new lines in *text* must also be hidden by backslashes. The **c** function 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.

No further commands are attempted on a line deleted by a **c** function.

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. See *Multiple Input-line Functions* for a description of the **r** function.

Note: Leading blanks and tabs are not displayed in the output produced by these functions. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash does not appear in the output.

For example, put the following list of editing commands in a file called *Xkubla*:

```
logo% cat > Xkubla
n
a\
XXXX
d
^D
logo% sed -f Xkubla kubla
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
logo%
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or

```
n
c\
XXXX
```

5.4.2. The Substitute Function 's'

The **s** (substitute) function changes parts of lines selected by a context search within the line. The standard format is the same as the *ed* substitute command:

*(2)***s** *pattern replacement flags*

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 *Specifying Lines for Input*. The only difference between *pattern* and a context address is that the context address must be delimited by slash ('/') characters; you can delimit *pattern* by any character other than space or newline.

By default, only the first string matched by *pattern* is replaced. 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* Is replaced by the *d*th substring matched by parts of *pattern* enclosed in '\(' and '\)' where *d* is a single digit. If nested substrings occur in *pattern*, the *d*th is determined

by counting opening delimiters ('\(').

As in patterns, you can make special characters 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 or 'display' the line if a successful replacement was done. The **p** flag writes the line 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 writes lines which are actually substituted by the **s** function 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**. You can specify a maximum of 10 different filenames after **w** flags and **w** functions (see below), combined.

For example, applying the following command to the the *kubla* file produces on the standard output:

```
logo% sed -e "s/to/by/w changes" kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

Note that if the edit command contains spaces, you must enclose it with quotes.

It also creates a new file called *changes* that contains only the lines changed as you can see using the *more* command:

```
logo% more changes
Through caverns measureless by man
Down by a sunless sea.
logo%
```

If the nocopy option **-n** is in effect, you see those lines that are changed:

```
logo% sed -e "s/[.,;?:]/*P&*/gp" -n kubla
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
logo%
```

Finally, to illustrate the effect of the **g** flag assuming nocopy mode, consider:

```
logo% sed -e "/X/s/an/AN/p" -n kubla
In XANadu did Kubla Khan
logo%
```

and the command:

```
logo% sed -e "/X/s/an/AN/gp" -n kubla
In XANadu did Kubla KhAN
logo%
```

5.4.3. Input-output Functions

The following functions affect the input and output of text. The maximum number of allowable addresses is in parentheses.

*(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. Put only one space between **w** and *filename*. ***You can use a maximum of ten different files in write functions and with 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 you execute **r** and **a** functions 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. Put only one space between 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 displayed.

Note: Since there is a limit to the number of files that can be opened simultaneously, put no more than ten files in **w** functions or flags; reduce that number by one if any **r** functions are present. Only one read file is open at one time.

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 reads in *note1* after the line containing 'Kubla':

```
logo% sed -e "/Kubla/r note1" kubla
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.
```

5.4.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with *pattern spaces* containing embedded newlines; they are intended principally to provide pattern matches across lines in the input. A pattern space is the range of pattern matches. 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** function described below.

The maximum number of allowable addresses is enclosed in parentheses.

*(2)***N** Next line. The next input line is appended to the current line in the pattern space; in embedded newline separates the two input lines. Pattern matches may extend across the embedded newline(s).

*(2)***D** Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

*(2)***P** Print or 'display' first part of the pattern space. Print up to and including the first newline in the pattern space.

The **P** and **D** functions are equivalent to their lower-case counterparts if there are no embedded newlines in the pattern space.

5.4.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

*(2)***h** Hold pattern space. The **h** function 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.

For example, if you want to add ':In Xanadu'to our standard example, create a file called *test* containing the following commands:

```
1h
1s/ did.*//
1x
G
s/\n/ :/
```

Then run that file on the *kubla* file:

```
logo% sed -f test kubla
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
logo%
```

5.4.6. Flow-of-Control Functions

These functions do not edit the input lines, but control the application of functions to the lines that are addressed.

*(2)***!** Called 'Don't', the '!' function applies the next command, written on the same line, to all and only those input lines *not* selected by the address part.

*(2)***{** Grouping. The grouping command '{' applies (or does not apply) the next set of commands as a block to the input lines that the addresses of the grouping command select. The first of the commands under control of the grouping command may appear on the same line as the '{' or on the next line.

A matching '}' standing on a line by itself terminates the group of commands. 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 restarts the sequence of editing commands being applied to the current input line 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. Either reading a new input line or executing a *t* function resets the flag which indicates that a successful substitution has occurred.

5.4.7. Miscellaneous Functions

Two additional functions are:

(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 writes the current line to the output if it should be, writes any appended or read text, and terminates execution.

Table of Contents

Chapter 6

Pattern Scanning and Processing with awk

Awk is a handy utility program, which you can program in varying degrees of complexity. *Awk*'s basic operation is to search a set of files for patterns based on *selection criteria*, and to perform specified actions on lines or groups of lines which contain those patterns. Selection criteria can be text patterns or *regular expressions*. *Awk* makes data selection, transformation operations, information retrieval and text manipulation easy to state and to perform.

Basic *awk* operation is to scan a set of input lines in order, searching for lines which match any of a set of patterns which you have specified. You can specify an action to be performed on each line that matches the pattern.

Awk patterns may include arbitrary Boolean combinations of regular expressions and of relational and arithmetic 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.

If you are familiar with the *grep* utility (see the *User's Manual for the Sun Workstation*), you 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 displaying the matching line.

As some simple examples to give you the idea, consider a short file called *sample*, which contains some identifying numbers and system names:

```
125.1303  krypton loghost
125.0x0733      window
125.1313  core
125.19          haley
```

If you want to display the second and first columns of information in that order, use the *awk* program:

```
logo% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
haley 125.19
```

This is good for reversing columns of tabular material for example. The next program shows all input lines with an a, b, or c in the second field.

```
logo% awk '$2 ~ /a|b|c/' sample
125.1313    core
125.19          haley
```

The material in this chapter is derived from *Awk — A Pattern Scanning and Processing Language*, A. Aho, B.W. Kernighan, P. Weinberger, Bell Laboratories, Murray Hill, New Jersey.

6.1. Using awk

The general format for using *awk* follows. You execute the *awk* commands in a string that we'll call *program* on the set of named *files*:

```
logo% awk program files
```

For example, to display all input lines whose length exceeds 13 characters, use the program:

```
logo% awk 'length > 13' sample
125.1303    krypton loghost
125.0x0733 window
logo%
```

In the above example, the *program* compares the length of the *sample* file lines to the number 13 and displays lines longer than 13 characters.

Awk usually takes its program as the first argument. To take a program from a file instead, use the **-f** (file) option. For example, you can put the same statement in a file called *howlong*, and execute it on *sample* with:

```
logo% awk -f howlong hosts
125.1303    krypton loghost
125.0x0733 window
```

You can also execute *awk* on the standard input if there are no files. Put single quotes around the *awk* program because the Shell duplicates most of *awk*'s special characters.

6.1.1. Program Structure

A program can consist of just an action to be performed on all lines in a file, as in the *howlong* example above. It can also contain a pattern that specifies the lines for the action to operate on. This pattern/action order is represented in *awk* notation by:

```
pattern    { action }
```

In other words, 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, the action is performed on every input line. A line which doesn't match any pattern is ignored. Since patterns and actions are both optional, you must enclose actions in braces ({action}) to distinguish them from patterns. See more about patterns in *Specifying Patterns*.

6.1.2. 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 separated by *field separators*, normally blanks or tabs, but you can change the input field separator, as described in *Field*

Separators. Fields are referred to as **$***X* where **$1** is the first field, **$2** the second, and so on as shown above. **$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; you can change them at any time to any single character. You may also use the optional command-line argument **–F***c* to set **FS** to any 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.

6.2. Displaying Text

The simplest action is to display (or *print*) some or all of a record with the *awk* command *print*. *Print* copies the input to the output intact. An action without a pattern is executed for all lines. To display each record of the *sample* file, use:

```
logo% awk '{print}' sample
125.1303    krypton loghost
125.0x0733 window
125.1313    core
125.19              haley
logo%
```

Remember to put single quotes around the *awk* program as we show here.

More useful than the above example is to print a field or fields from each record. For instance, to display the first two fields in reverse order, type:

```
logo% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
logo%
```

Items separated by a comma in the *print* statement are separated by the current output field separator when output. Items not separated by commas are concatenated, so to run the first and second fields together, type:

```
logo% awk '{print $1 $2}' sample
125.1303krypton
125.0x0733window
125.1313core
125.19haley
logo%
```

You can use the predefined variables **NF** and **NR**; for example, to print each record preceded by the record number and the number of fields, use:

```
logo% awk '{ print NR, NF, $0 }' sample
1 3 125.1303      krypton loghost
2 2 125.0x0733    window
3 2 125.1313      core
4 2 125.19        haley
logo%
```

You may divert output to multiple files; the program:

logo%**awk '{print $1 >"foo1"; print $2 >"foo2" }'** *filename*

writes the first field, **$1**, on the file *foo1*, and the second field on file *foo2*. You can also use the '>>' notation; to append the output to the file *foo* for example, say:

logo% **awk '{print $1 >>"foo"}'** *filename*

In each case, the output files are created if necessary. The filename can be a variable or a field as well as a constant. For example, to use the contents of field 2 as a filename, type:

logo% **awk '{print $1 >$2}'** *filename*

This program prints the contents of field 1 of *filename* on field 2. If you run this on our *sample* file, four new files are created. There is a limit of 10 output files.

Similarly, you can pipe output into another process. For instance, to mail the output of an *awk* program to *henry*, use:

logo% **awk '{ print NR, NF, $0 }' sample | mail henry**

(See the *Mail User's Guide* in the *Beginner's Guide to the Sun Workstation* for details on *mail*.)

To change the current output field separator and output record separator, use the variables **OFS** and **ORS**. The output record separator is appended to the output of the *print* statement.

Awk also provides the *printf* statement for output formatting. To format the expressions in the list according to the specification in *format* and print them, use:

logo% **awk printf** *format, expr, expr, ...*

To print **$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, use:

logo% **awk '{printf("%8.2f %10ld\n",$1,$2)}'** *filename*

Notice that you have to specifically insert spaces or tab characters by enclosing them in quoted strings. Otherwise, the output appears all scrunched together. The version of *printf* is identical to that provided in the C Standard I/O library (see *printf* in *C Library Standard I/O* (3S) in the *System Interface Manual for the Sun Workstation*).

6.3. Specifying Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. You may use a variety of expressions as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary Boolean combinations of these.

6.3.1. BEGIN and END

Awk has two built-in patterns, BEGIN and END . 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.

6.3.2. 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 displays all lines which contain any occurrence of the name 'smith'. If a line contains 'smith' as part of a larger word, it is also displayed. Suppose you have a file *testfile* that contains:

```
summertime
smith
blacksmithing
Smithsonian
hammersmith
```

If you use *awk* on it, the display is:

```
logo% awk /smith/ testfile
smith
blacksmithing
hammersmith
```

Awk regular expressions include the regular expression forms found in the text editor *ed* and in *grep* (see the *User's Manual for the Sun Workstation).* In addition, *awk* uses parentheses for grouping, '|' for alternatives, '+ ' for 'one or more', and '?' for 'zero or one', all as in *lex.* Character classes may be abbreviated. For example:

```
/[a-zA-Z0-9]/
```

is the set of all letters and digits. As an example, to display all lines which contain any of the names 'Adams,' 'West' or 'Smith,' whether capitalized or not, use:

```
'/[Aa]dams|[Ww]est|[Ss]mith/'
```

Enclose regular expressions (with the extensions listed above) in slashes, just as in *ed* and *sed.* For example:

```
logo% awk '/[Ss]mith/' testfile
smith
blacksmithing
Smithsonian
hammersmith
```

finds both 'smith' and 'Smith'.

Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off 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.

Use the operators '~' and '!~' to find if any field or variable matches a regular expression (or does not match it). The program

```
$1 ~ /[sS]mith/
```

displays all lines where the first field matches 'smith' or 'Smith.' Notice that this will also match 'blacksmithing', 'Smithsonian', and so on. To restrict it to exactly [sS]mith, use:

```
logo% awk '$1 ~ /^[sS]mith$/' testfile
smith
logo%
```

The caret '^' refers to the beginning of a line or field; the dollar sign '$' refers to the end.

6.3.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational and arithmetic operators ' < ', '<= ', ' == ', ' != ', ' >= ', and ' > ', the same as those in C. An example is:

```
'$2 > $1 + 100'
```

which selects lines where the second field is at least 100 greater than the first field.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
logo% awk '$1 >= "s"' testfile
smith
```

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
```

performs a string comparison between field 1 and field 2.

6.3.4. Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators ' || ' (or), ' && ' (and), and ' ! ' (not). For example, to select lines where the first field begins with 's', but is not 'smith', use:

```
logo% awk '$1 >= "s" && $1 < "t" && $1 != "smith"' testfile
summertime
```

' && ' and ' || ' guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

The program:

```
$1 !=prev {print; prev=$1}
```

displays all lines in which the first field is different from the previous first field.

6.3.5. Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma, as in

```
pattern1, pattern2    { ... }
```

In this case, the action is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* inclusive. For example, to display all lines between the strings 'sum' and 'black', use:

```
logo% awk '/sum/, /black/' testfile
summertime
smith
blacksmithing
logo%
```

while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

6.4. 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.

6.4.1. Assignments, Variables, and Expressions

The simplest action is an *assignment.* For example, you can assign 1 to the *variable x*:

```
x = 1
```

The '1' is a simple *expression. Awk* variables can take on numeric (floating point) or string values according to context. 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, to assign 7 to *x*, use:

```
x = "3" + "4"
```

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). For example:

```
NF % 2 == 0
```

displays lines with an even number of fields. To display all lines with an even number of fields, use:

```
NF % 2 == 0
```

The C increment '++' and decrement '--' operators are also available, and so are the assignment operators '+=', '-=', '*=', '/=', and '%='.

An *awk* pattern can be a *conditional expression* as well as a simple expression as in the 'x = 1' assignment above. The operators listed above may all be used in expressions. An *awk* program with a conditional expression specifies conditional selection based on properties of the individual fields in the record.

6.4.2. 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.

To replace the first field of each line by its logarithm, say:

```
{ $1 = log($1); print }
```

Thus you 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 considered numeric or string depends on context; fields are treated as strings

in ambiguous cases like:

```
if ($1 == $2) ...
```

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields. To split the string 's' into 'array[1]' ..., 'array[n]', use:

```
n = split(s, array, sep)
```

This returns the number of elements found. If the **sep** argument is provided, it is used as the field separator; otherwise **FS** is used as the separator.

6.4.3. 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.

6.4.4. Built-in Functions

Awk provides several *built-in* functions.

6.4.4.1. Length Function

The 'length' function computes the length of a string of characters. This program shows each record, preceded by its length:

```
logo% awk '{print length, $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
logo%
```

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:

```
logo% awk '{print length($0), $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
```

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
```

displays lines whose length is less than 10 or greater than 20.

6.4.4.2. Substring Function

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*.

6.4.4.3. Index Function

The function *index(s1, s2)* returns the position where the string *s2* occurs in *s1*, or zero if it does not.

6.4.4.4. Sprintf Function

The function *sprintf(f, e1, e2, ...)* produces the value of the expressions *e1*, *e2*, and so on, in the *printf* format specified by *f*. Thus, for example, to set x to the string produced by formatting the values of $1 and $2, use:

```
x = sprintf("%8.2f %10ld", $1, $2)
```

6.4.5. 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.

6.4.6. 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 *Field Variables* 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.

You may put comments in *awk* programs: begin them with the character '#' and end them with the end of the line, as in

```
print x, y   # this is a comment
```

Table of Contents

Table of Contents

List of Tables

Chapter 1

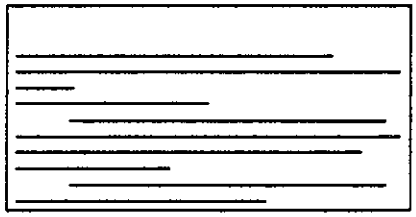
Introduction to Document Preparation

The main document preparation programs in the Sun System are *nroff* and *troff*. These programs handle one or more files containing both the text to be formatted and requests specifying how the output should look. From this input, the programs produce formatted output: *nroff* on typewriter-like terminals, and *troff* on a phototypesetter. Although they are separate programs, *nroff* and *troff* are compatible; they share the same command language and produce their output from the same input file. Descriptions here apply to both *troff* and *nroff* unless indicated otherwise.

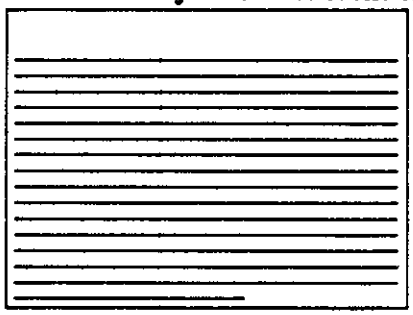
1.1. What Do Text Formatters Do?

You can type in the text of a document on lines of any length, and the text formatters produce lines of uniform length in the finished document. This is called *filling*, which means that the formatter collects words from what you type in the input file, and places them on an output line until no more will fit within a given line length. It *hyphenates* words automatically, so a line may be completed with part of a word to produce the right line length. It also *adjusts* a line after it has been filled by inserting spaces between words as necessary to bring the text exactly to the right margin. Examples of filling and adjusting follow:

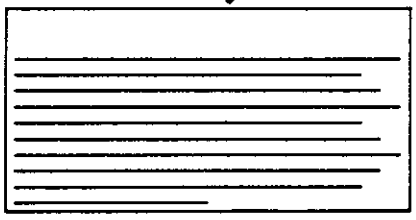
Unfilled text looks like:



Filled and adjusted text looks like:



Filled but not adjusted text looks like:



Given a file of input consisting only of lines of text without any formatting requests, the formatter simply produces a continuous stream of filled, adjusted and hyphenated output.

To obtain paragraphs, numbered sections, multiple column layout, tops and bottoms of pages, and footnotes, for example, requires the addition of formatting requests. Requests look like '*.xx*' where *xx* is one or two lower-case letters or a lower-case letter and a digit. Refer to *Formatting Documents with 'nroff' and 'troff'* for details.

1.2. What is a Macro Package?

Nroff and *troff* provide a flexible, sophisticated command language for requesting operations like those just mentioned. They are very flexible, but this flexibility can make them difficult to use because you have to use several requests to produce a simple format. For this reason, it's a good idea to use a *macro package.* A macro is simply a *predefined sequence of nroff requests or text* which you can use by including just one request in your input file. You can then handle repetitious tasks, such as starting paragraphs and numbering pages, by typing one macro request each time instead of several. A macro looks like '*.XX*' where *XX* is one or two upper-case letters or an upper-case letter and a digit.

A macro package also does a lot of things without the instructions that you have to give *nroff,* footnotes and page transitions for example. Some packages set up a page layout style by default, but you can change that style if you wish. Although a macro package offers only a limited subset of the wide range of formatting possibilities that *nroff* provides, it is much easier to use. We explain how to use a macro package in conjunction with *nroff* and *troff* in *Displaying and Printing Documents.*

Sample input with both formatting requests, macros in this case, and text looks like:

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

Refer to *Formatting Documents with the -ms Macros* and to *Quick References* in this chapter for more information on macros.

1.3. What is a Preprocessor?

A *preprocessor* is a program that you run your text file through first before passing it on to a text formatter. You can put tables in a document by *preprocessing* a file with the table-builder called *tbl.* You can add mathematical equations with their special fonts and symbols with the equation formatters, *eqn* for *troff* files and *neqn* for *nroff* files. These preprocessors convert material entered in their specific command languages to straight *troff* or *nroff* input. Those text formatters then produce the tables or mathematical equations for the output.

What you type in a file is very much the same as for simple formatting. You include table or equation material in your *troff* input file along with ordinary text and add several specific *tbl* or *eqn* requests. Refer to *Formatting Tables with 'tbl'* and *Formatting Mathematics with 'eqn'* for details.

1.4. Typesetting Jargon

There are several printer's measurement terms that are borrowed from traditional typesetting. These terms describe the size of the letters, the distance between lines and paragraphs, how long each line is, where the text is placed on the page, and so on.

Point *Points* specify the *size* of a letter or *type*. A point measures about 1/72 of an inch, which means that there are 72 points to the inch. This manual is in 10-point type, for instance.

Ems and *Ens*

Ems and *ens* are measures of distance and are proportional to the type size being used. An *em* is the distance equal to the number of points in the width of the letter 'm' in that point size. For examples, here's an em in several point sizes followed by an em dash to show why this is a *proportional* unit of measure. You wouldn't want a 20-point dash if you are printing the rest of a document in 12-point. Here's 12-point:

And here's 20-point:

An *en* space is one half of an *em* or about the width of the letter 'n'. They are typically used for indicating indention.

Vertical Spacing

Vertical spacing called *leading* (pronounced 'led-ing') is the distance between the bottom of one line and the bottom of the next. This manual has 12-point vertical spacing for example. The rule of thumb is that the spacing be 20% bigger than the character size for easy readability. A printer would call the ratio for this manual "Ten over twelve."

Paragraph Depth

As there is a specification for the distance between lines, there is also a term for the space between paragraphs. This is the *paragraph depth*. If you are using the standard '.PP' macro, for instance, the paragraph depth is whatever one vertical space has been set to.

Paragraph Indent

This is the amount of space that the first line is indented in relation to the rest of the paragraph. If you use a '.PP' macro to format a standard indented paragraph, the indent is two em spaces as shown by the first line in this paragraph.

Line Length

Line length specifies the width of text on a page. Here we use a six-inch line length. Shortening the line length generally makes text easier to read. Recall that many magazines and newspapers have 2-1/4 inch columns for quick reading.

Page Offset

Page offset determines the left margin, that is how far in from the left edge of the paper the text is set. On a normal 8-1/2 by 11 letter-size page, the page offset is is normally slightly less than one inch.

Indent The *indent* of text is the distance the text is set in from the page offset. The indent emphasizes the text by setting it off from the rest.

1.5. Hints for Typing in Text

The following provides a few tricks for typing in text and for further online editing and formatting.

- A period (.) or apostrophe (') as the first character on a line indicates that the line contains a formatting request. If you type a line of text beginning with either of these *control characters*, *nroff* tries to interpret them as a request, and the rest of the text on that line disappears. If you *have* to print a period or an apostrophe as the first character, escape their normal meanings by prefixing them with a backslash and an ampersand, \&..., for instance.
- Following the control character is a one- or two-character *name* of a formatting request. As described earlier, *nroff* and *troff* names usually consist of one or two lower-case letters or a lower-case letter and a digit. Macro package names usually consist of one or two upper-case letters or one upper-case letter and a digit. For example, '.sp' is an *nroff* request for a space and '.PP' is an *-ms* macro request for an indented paragraph.
- End a line of text with the end of a word along with any trailing punctuation. *Nroff* inserts a space between whatever ends one line of input text and whatever begins the next.
- Start lines in the input file with something other than a space. A space at the beginning of an input line creates a *break* at that point in the output and *nroff* skips to a new output line, interrupting the process of filling and adjusting. This is the easiest way to get spaces between paragraphs, but it does not leave much flexibility for changing things later.
- Some requests go on a line by themselves, while others can take one or more additional pieces of information on the same line. These extra pieces of information on the request line are called *arguments*. Separate them from the request name and from each other by one or more spaces. Sometimes the argument is a piece of text on which the request operates; other times it can be some additional information about what the request is to do. For example, the vertical space request '.sp 3' shows an *nroff* request with one argument. It requests three blank spaces.

1.6. Types of Paragraphs

There are several types of paragraphs. When should you use one type of paragraph instead of another? Here are a few words about paragraphs, their characteristics, and formatting in general. See the *Types of Paragraphs* figure that follows for examples.

Use regular indented and block paragraphs for narrative descriptions. It is a matter of style as to which type you choose to use. In general, indented paragraphs remove the need for extra space between paragraphs — the indent tells you where the start of the new paragraph is. Most business communication is done with block paragraphs.

If you want to indicate a set of points without any specific order, use a bulleted list. For example:

There are many kinds of coffee:

- Jamaica Blue Mountain
- Colombian
- Java
- Moka
- French Roast
- Major Dickenson's Blend

When you want to describe a set of things in some order, such as a step-by-step procedure, use a numbered list:

To repair television, follow these steps:

1. Remove screws in rear casing.
2. Carefully slide out picture tube.
3. Gently smash with hammer.

Use description lists to explain a set of related or unrelated things, or sometimes to highlight keywords. For instance,

Options

-v	Verbose
-f *filename*	Take script from *filename*
-o	Use old format

In typographic parlance, anything that is not part of the "body text" — regular paragraphs and such — is considered a ***display***, and often has to be specially handled. Generally a display is "displayed" exactly as you type it or draw it originally, with no interference from the formatter. Displays are used to set off important text, special effects, drawings, or examples, as we do throughout this manual, The following paragraph is a ***display***.

Tom appeared on the sidewalk with a bucket of whitewash and
a long-handled brush.
He surveyed the fence, and all gladness left him and
a deep melancholy settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.

Quotations set off quoted material from the rest of the text for emphasis. For example,

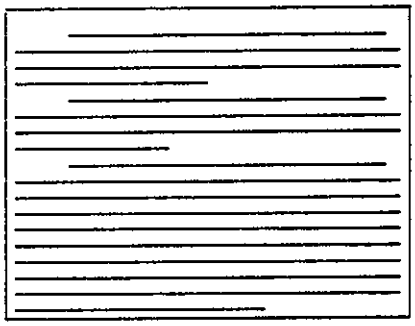
"... in the conversation between Alice and the Queen, we read this piece of homespun philosophy:

> "A slow sort of country!" said the Queen. "Now, ***here***, you see, it takes all the running you can do, to keep in the same place. If you want to get somethwere else, you must run at least twice as fast as that!"
>
> ***Through the Looking Glass***
> Lewis Carroll

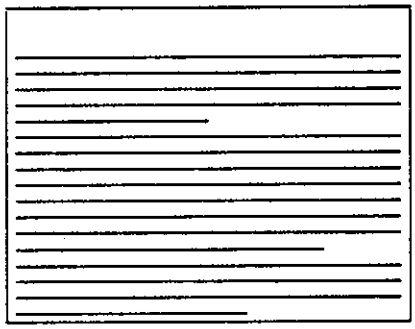
Examine the following thumbnail sketches of paragraph types to see how each can serve a special function:

Table 1-1: Types of Paragraphs

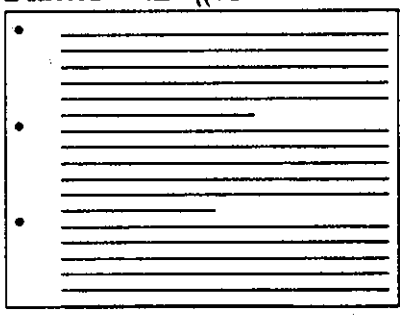
Indented — .PP



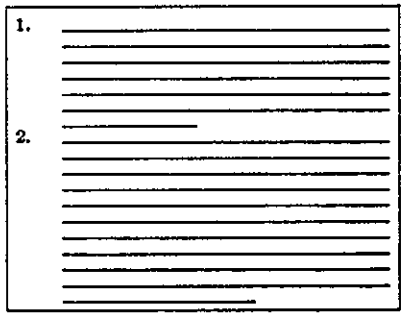
Left Block — .LP



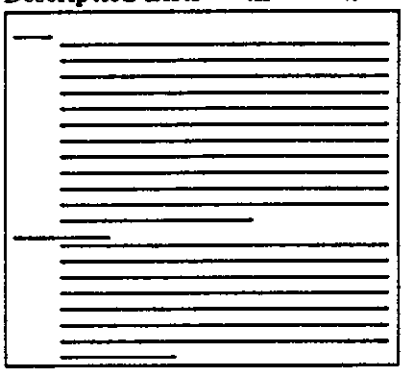
Bulleted — .IP \(bu



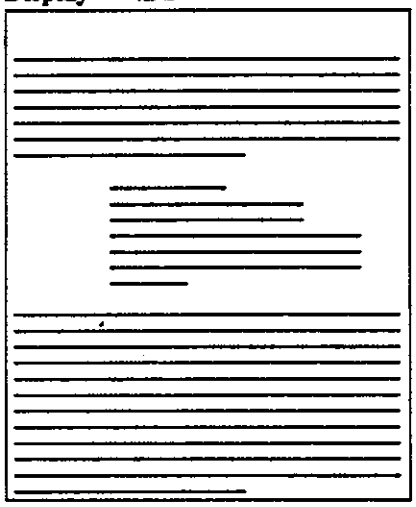
Numbered — .IP 1.



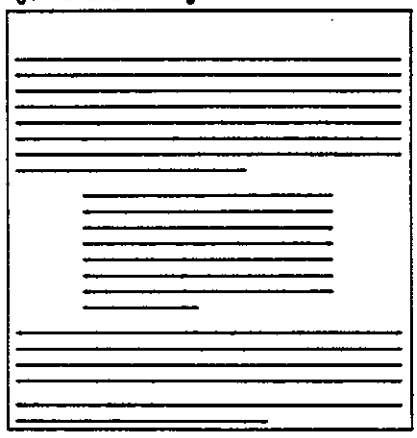
Description Lists — .IP " " *n*



Display — .DS



Quotation — .QP



1.7. Quick References

This section[1] provides some simple templates for producing your documents with the **–ms** macro package. Remember that for a quick, paginated, and justified document, you can simply type an '.LP' to start your document, and then type in the text separated by blank lines to produce paragraphs. Type a space and RETURN to get a blank line.

Throughout the examples, input is shown in

bold Times Roman font

while the output is shown in

this Times Roman font.

1.7.1. Displaying and Printing Documents

Use the following to format and print your documents. You can use either *nroff* or *troff* depending on the output you desire. Use *nroff* to either display formatted output on your workstation screen or to print a formatted document. The default is to display on the standard output, your workstation screen. For easy viewing, pipe your output to *more* or redirect the output to a file.

Using *troff* or your installation's equivalent prepares your output for phototypesetting.

[1] Some of the material in this section is derived from *A Guide to Preparing Documents with '–me'*, M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey.

Table 1-2: How to Display and Print Documents

What You Want to Do	How to Do It
Display simple text	**nroff** *–options files*
Display text with tables only	**tbl** *files* | **nroff** *–options*
Display text with equations only	**neqn** *files* | **nroff** *–options*
Display text with both tables and equations	**tbl** *files* | **neqn** | **nroff** *–options*
Print raw text and requests	**pr** *files* | **lpr** *–Pprinter*
Print text	**nroff** *–options files* | **lpr** *–Pprinter*
Print text with tables only	**tbl** *files* | **nroff** *–options* | **lpr** *–Pprinter*
Print text with equations only	**neqn** *files* | **nroff** *–options* | **lpr** *–Pprinter*
Print text with both tables and equations	**tbl** *files* | **neqn** | **nroff** *–options* | **lpr** *–Pprinter*
Phototypeset simple text	**troff** *–options files*
Phototypeset text with tables	**tbl** *files* | **troff** *–options*
Phototypeset text with equations	**eqn** *files* | **troff** *–options*
Phototypeset text with both tables and equations	**tbl** *files* | **eqn** | **troff** *–options*

1.7.2. Technical Memorandum

Here we provide a sample format for a technical memorandum.

Input:

```
.DA March 11, 1983
.TL
An Analysis of
Cucumbers and Pickles
.AU
A. B. Hacker
.AU
C. D. Wizard
.AI
Stanford University
Stanford, California
.AB
This abstract should be short enough to
fit on a single page cover sheet.
It provides a summary of memorandum
contents.
.AE
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.NH
References
```

Ouput:

An Analysis of
Cucumbers and Pickles

A. B. Hacker
C. D. Wizard

Stanford University
Stanford, California

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It provides a summary of memorandum contents.

1. Introduction.

Now the first paragraph of actual text ...

1. References

1.7.3. Section Headings for Documents

.NH **Introduction.** **.PP** *text text text*	**.SH** **Appendix I** **.PP** *text text text*
1. Introduction text text text	Appendix I text text text

1.7.4. Changing Fonts

The following table shows the easiest way to change the default roman font to italic or bold. To change the font of a single word, put the word on the same line as the macro. To change more than one word, put them on the line following the macro.

Input	Output
.I Hello	*Hello*
.I **Puts this line in italics.**	*Puts this line in italics.*
.B Goodbye	**Goodbye**
.B **Prints this line in bold.**	**Prints this line in bold.**
.R **Prints this line in roman.**	Puts this line in roman.

1.7.5. Making a Simple List

Use the following template for a simple list.

Input:

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

Output:

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

1.7.6. Multiple Indents for Lists and Outlines

This template shows how to format lists or outlines.

Input:

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

Output:

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

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

1.7.7. Displays

A display does not fill or justify the text. It keeps the text together, and sets the lines off from the rest.
Input:

```
text text text text text text
.DS
and now
for something
completely different
.DE
text text text text text text
```

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

and now
for something
completely different

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

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

1.7.8. Footnotes

For automatically numbered footnotes, put the predefined string ** at the end of the text you want to footnote like this:[2]

```
you want to footnote like this:\**
.FS
Here's a numbered footnote.
.FE
```

To mark footnotes with other symbols, put the symbol as the first argument to '.FS' and at the end of the text you want to footnote like this:†

```
and at the end of the text you want tot footnote like this:\(dg
.FS \(dg
You can also use an asterisk (*) or a double dagger ‡ (\(dd).
.FE
```

1.7.9. Keeping Text Together — Keeps

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

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

[2] Here's a numbered footnote.

† You can also use an asterisk (*) or a double dagger ‡ (\(dd).

1.7.10. Double-Column Format

Put a '.2C' at the beginning of the material you want printed in two columns. To return to one column, use '.1C'. Note that '.1C' breaks to a new page.

Input:

```
.TL
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of ...
```

1.7.11. Sample Tables

Two sample table templates follow.

Input:

```
.TS
box center tab (/);
lB lB
l l.
Column Header Column Header
_
text/text
text/text
text/text
text/text
.TE
```

Output:

Column Header	Column Header
text	text
text	text
text	text
text	text

Input:

```
.TS
allbox tab (/);
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)

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

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

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

Input:

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

Output:

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

1.7.12. Writing Mathematical Equations

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

Input:

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

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

Output:

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

Input:

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

Output:

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

Input:

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

Output:

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

$$= \left[\frac{\partial V}{\partial x}\right]^2 + \left[\frac{\partial V}{\partial y}\right]^2 \qquad \lambda \rightarrow \infty$$

Input:

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

Output:

$\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$.

(with delim $$ on).

1.7.13. Registers You Can Change

Table 1-3: Registers You Can Change

Controls	Register	Controls	Register
Line length	.nr LL 7i	Title length	.nr LT 7i
Point size	.nr PS 9	Vertical spacing	.nr VS 11
Column width	.nr CW 3i	Intercolumn spacing	.nr GW .5i
Margins – head and foot	.nr HM .75i .nr FM .75i	Paragraph indent	.nr PI 2n
Paragraph spacing	.nr PD 0	Page offset	.nr PO 0.5i
Page heading	.ds CH Appendix (center) ds RH 7-25-76 (right) .ds LH Private (left)	Page footer	.ds CF Draft .ds LF .ds RF similar
Page numbers	.nr % 3		

Table of Contents

List of Tables

Chapter 2

Formatting Documents with the –ms Macros

This chapter[1] describes the *new -ms* macro package for preparing documents with *nroff* and *troff* on the Sun system. The *-ms Request Summary* at the end of this chapter provides a quick reference for all the –ms macros and for useful displaying and printing commands. If you are acquainted with –ms, there is a quick reference for the *new* requests and string definitions as well. The differences between the new and the old –ms macro packages are described in *Changes in the New '-ms' Macro Package. Displaying and Printing with '-ms'* describes how you can produce documents on either your workstation, or on the printer or phototypesetter without changing the text and formatting request input.

2.1. Changes in the New –ms Macro Package

The old –ms macro package has been revised, and the new macro package assumes the name –ms. There are some extensions to previous –ms macros and a number of new macros, but all the previously documented –ms macros still work exactly as they did before, and have the same names as before. The new –ms macro package includes several bug fixes, including a problem with the single-column '.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* bibliographic citation macros. Macros used only at Bell Laboratories have been removed from the new version. We list them at the end of this chapter in *-ms Request Summary.*

Note: The new macros are annotated with ‡.

2.2. Displaying and Printing Documents with –ms

After you have prepared your document with text and *-ms* formatting requests and stored it in a file, you can display it on your workstation screen or print it with *nroff* or *troff* with the –ms option to use the *-ms* macro package. A good way to start is to pipe your file through *more* for viewing:

logo% **nroff –ms** *file* ... **| more**

If you forget the *-ms* option, you get continuous, justified, unpaginated output in which *-ms* requests are ignored. You can format more than one file on the command line at a time, in which case *nroff* simply processes all of them in the order they appear, as if they were one file. There are other *options* to use with *nroff* and *troff*; see the *User's Manual for the Sun Workstation* for details.

[1] The material in this chapter is derived from *A Revised Version of '-ms'*, B. Tuthill, University of California, Berkeley; *Typing Documents on the UNIX System: Using the '-ms' Macros with* 'troff' and 'nroff', M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey; and *Document Formatting on UNIX: Using the '-ms' Macros*, Joel Kies, University of California, Berkeley.

You can get preview and final output of various sorts with the following commands. To send *nroff* output to the line printer, type:

logo% **nroff -ms** *file* **| lpr** *-printer*

To produce a file with tables, use:

logo% **tbl** *file* **| nroff -ms | lpr** *-printer*

To produce a file with equations, type:

logo% **neqn** *file* **| nroff -ms | lpr** *-printer*

To produce a file with tables and equations, use the following order:

logo% **tbl** *file* **| neqn | nroff -ms | lpr** *-printer*

To print your document with *troff*, use:

logo% **troff -ms** *file* **| lpr -t** *-printer*

Use the same order with *troff* for preprocessing files with *tbl* and *eqn*.

If you use the two-column '.2C' request, either pipe the *nroff* output through *col* or make the first line of the input '.pi /usr/bin/col.'

See *lpr* in the *User's Manual for the Sun Workstation* for details on printing.

2.3. What Can Macros Do?

Macros can format facilities for paragraphs, lists, sections (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, a table of contents, endnotes, running heads and feet, and cover pages for papers. As with other formatting utilities such as *nroff* and *troff*, you prepare text interspersed with formatting requests. However, the macro package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. In other words, you can do a lot more done with just one macro than with one *nroff* request.

2.4. Formatting Requests

When you use a macro package, you type in text as you normally do and intersperse it with formatting *requests*. For example, instead of spacing in with the space bar or typing a tab to indent for paragraphs, type a line with the '.PP' request before each paragraph. When formatted, this leaves a space and indents the first line of the following paragraph.

An *-ms* request is one or two upper-case characters, and usually in the form '.*XX*'.

The easiet way to produce simple formatted text is to put an '.LP' request at the start of the document and add your text, typing just a space to separate paragraphs. The '.LP' produces a left-aligned (block) paragraph, as used throughout this chapter. Your output will have paragraphs and be paginated with right and left-justified margins.

Note: You cannot just begin a document with a line of text. You must include some *-ms* request before any text input. When in doubt, use '.LP' to properly *initialize* the file, although any of the requests '.PP', '.LP', '.TL', '.SH', '.NH' is good enough. See *Cover Sheets and Title Pages* for the correct arrangement of requests at the start of a document.

2.4.1. Paragraphs

You can produce several different kinds of paragraphs with the *-ms* macro package: standard, left-block, indented, labeled, and quoted.

2.4.1.1. Standard Paragraph — '.PP'

To get an ordinary paragraph, use the '.PP' request, followed on subsequent lines by the text of the paragraph. For example, you type:

```
.PP
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.
```

to produce:

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush. He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

2.4.1.2. Left-Block Paragraph — '.LP'

You can also produce a left-block paragraph, like those in this manual, with '.LP'. The first line is not indented as it is with the '.PP' request. For example, you type:

```
.LP
Tom appeared ...
```

to produce:

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush. He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

There are default values for the vertical spacing before paragraphs and for the width of the indentation. To change the paragraph spacing, see *Modifying Default Features*.

2.4.1.3. Indented Paragraph — '.IP'

Another kind of paragraph is the indented paragraph, produced by the '.IP' request. These paragraphs can have hanging numbers or labels. For example:

```
.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, ...
.LP
```

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. Here we used the '.LP' request.

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

```
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
.IP
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.
.LP
```

which produces

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush.

He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

If a non-standard amount of indenting is required, specify it after the label in character positions. It remains in effect until the next '.PP' or '.LP'. Thus, the general form of the '.IP' request contains two additional fields: the label and the indenting length. For example,

```
.IP "Example one:" 15
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP "Example two:"
And so forth.
.LP
```

produces this:

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

Example two: And so forth.

Notice that you must enclose the label in double quote marks because it contains a space; otherwise, the space signifies the end of the argument. The indentation request above is in the number of *ens*, a unit of dimension used in typesetting. An *en* is approximately the width of a lowercase 'n' in the particular point size you are using.

The '.IP' macro adjusts properly by causing a break to the next line if you type in a label longer than the space you allowed for. For example, if you have a very long label and have allowed 10 n spaces for it, your input looks like:

```
.IP "A very, very, long and verbose label" 10
And now here's the text that you want.
And now here's the text that you want.
And now here's the text that you want.
And now here's the text that you want.
And now here's the text that you want.
```

And your output is adjusted accordingly with a break between the label and the text body:

A very, very, long and verbose label
And now here's the text that you want. And now here's the text that you want. And now here's the text that you want. And now here's the text that you want. And now here's the text that you want.

2.4.1.4. Nested Indentation — '.RS' and '.RE'

It is also possible to produce multiple (or *relative*) nested indents; the '.RS' request indicates that the next '.IP' starts its indentation from the current indentation level. Each '.RE' undoes one level of indenting, so you should balance '.RS' and '.RE' requests. Think of the '.RS' request as 'move right' and the '.RE' request as 'move left'. As an example:

```
.IP I.
South Bay Area Restaurants
.RS
.IP A.
Palo Alto
.RS
.IP 1.
La Terrasse
.RE
.IP B.
Mountain View
.RS
.IP 1.
Grand China
.RE
.IP C.
Menlo Park
.RS
.IP 1.
Late for the Train
.IP 2.
Flea Street Cafe
.RE
.RE
.LP
```

results in

I. South Bay Area Restaurants
   A. Palo Alto
      1. La Terrasse
   B. Mountain View
      1. Grand China
   C. Menlo Park
      1. Late for the Train
      2. Flea Street Cafe

Note the two '.RE' requests in a row at the end of the list. Remember that you need one *end* for each *start*.

2.4.1.5. Quoted Paragraph — '.QP'

All of the variations on '.LP' leave the right margin untouched. Sometimes, you need a a paragraph indented on both right and left sides. To set off a quotation as such, use:

```
.QP
Precede each paragraph that you
want offset as a quotation
with a '.QP'. This produces a paragraph
like this.
Notice that the right edge is also indented from the right margin.
```

to produce

> Precede each paragraph that you want offset as a quotation with a '.QP'. This produces a paragraph like this. Notice that the right edge is also indented from the right margin.

2.4.2. Section Headings — '.SH' and '.NH'

There are two varieties of section headings, unnumbered with '.SH' and numbered with '.NH'. In either case, type the text of the section heading on one or more lines following the request. End the section heading by typing a subsequent paragraph request or another section heading request. When printed, one line of vertical space precedes the heading, which begins at the left margin. *Nroff* offsets the heading with blank lines, while *troff* sets it in **boldface** type. '.NH' section headings are numbered automatically. The macro takes an argument number representing the *level-number* of the heading, up to 5. A third-level section number is one like '1.2.1'. The macro adds one to the section number at the requested level, as shown in the following example:

```
.NH
Bay Area Recreation
.NH 2
Beaches
.NH 3
San Gregorio
.NH 3
Half Moon Bay
.NH 2
Parks
.NH 3
Wunderlich
.NH 3
Los Trancos
.NH 2
Amusement Parks
.NH 3
Marine World/Africa USA
```

generates:

2. Bay Area Recreation

2.1 Beaches

2.1.1 San Gregorio

2.1.2 Half Moon Bay

2.2 Parks

2.2.1 Wunderlich

2.2.2 Los Trancos

2.3 Amusement Parks

2.3.1 Marine World/Africa USA

'.NH' without a level-number means the same thing as '.NH 1', and '.NH 0' cancels the numbering sequence in effect and produces a section heading numbered 1.

2.4.3. Cover Sheets and Title Pages

–Ms provides a group of macros to format items that typically appear on the cover sheet or title page of a formally laid-out paper. You can use them selectively, but if you use several, you must put them in the order shown below, normally at or near the beginning of the input file.

The first line of a document signals the general format of the first page. In particular, if it is '.RP' (released paper), a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

Sample input is:

.RP *(Optional for released paper format)*
.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')

(See *Order of Requests in Input* for a quick example of this scheme.)

If the '.RP' request precedes '.TL', the title, author, and abstract material are printed separately on a cover sheet. The title and author information (not the abstract) is then repeated automatically on page one (the title page) of the paper, without your having to type it again. If you do not include an '.RP' request, all of this material appears on page one, followed on the same page by the main text of the paper.

To omit some of the standard headings (such as no abstract, or no author's institution), just omit the corresponding fields and command lines. To suppress the word ABSTRACT type '.AB no' for '.AB'. You can intersperse several '.AU' and '.AI lines to format for multiple authors.

These macros are optional; you may begin a paper simply with a section heading or paragraph request. When you do precede the main text with cover sheet and title page material, include a paragraph or section heading between the last title page request and the beginning of the main text. Don't forget that some *-ms* request must precede any input text.

2.4.4. Running Heads and Feet — 'LH', 'CH', 'RH'

The *-ms* macros, by default, print a page heading containing a page number (if greater than 1). You can make minor adjustments to the page headings and 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 *nroff* output, there are two default values: CH is the current page number surrounded on either side by hyphens, and CF contains the current date as supplied by the computer. For *troff* CH also contains the page number, but CF is empty. The other four registers are empty by default for both *nroff* and *troff*. You can use the '.ds' request to assign a value to a string register. For example:

```
.ds RF Draft Only \(em Do Not Distribute
```

This prints the character string 'Draft Only — Do Not Distribute' at the bottom right of every page. You do not need to enclose the string in double quote marks. To remove the contents of a string register, simply redefine it as empty. For instance, to clear string register CH, and make the center header blank on the following pages, use the request:

```
.ds CH
```

To put the page number in the right header, use:

```
.ds RH %
```

In a string definition, '%' is a special symbol referring to *nroff*'s automatic page counter. If you want hyphens on either side of the page number, place them on either side of the '%' in the command, that is:

```
.ds RH -%-
```

Remember that putting the page number in the right header as shown above does not remove it from the default CF; you still have to clear out CF.

If you want requests that set the values of string and number registers to take effect on the first page of output, put them at or near the beginning of the input file, before the initializing macro, which in turn must precede the first line of text. Among other functions, the initializing macro causes a 'pseudo page break' onto page one of the paper, including the top-of-page processing for that page. Be sure to put requests that change the value of the PO (page offset), HM (top or head margin), and FM (bottom or foot margin) number registers and the page header string registers before the transition onto the page where they are to take effect.

For more complex formats, you can redefine the macros PT (page top) and BT (page bottom), which are invoked respectively at the top and bottom of each page. The margins (taken from registers HM and FM for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. If you redefine these macros, be careful not to change parameters such as point size or font without resetting them to default values.

2.4.5. Custom Headers and Footers — '.OH', '.EH', '.OF', and 'EF'

You can also 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 the *nroff* '.tl', that is, there are three fields (left, center and right), each separated by a single apostrophe. For example, to get odd-page headers with the chapter name followed by the page number and the reverse on even pages, use:

```
.OH 'For Whom the Bell Tolls''Page %'
.EH 'Page %''For Whom the Bell Tolls'
```

if you need one, 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'.

You can use the '.P1'‡ (P one) macro to print the header on page 1. If you want roman numeral page numbering, use an '.af PN i' request.

2.4.6. Multi-column Formats — '.2C' and '.MC'

If you place the request '.2C' in your document, the document will be printed in double column format beginning at that point. This is often desirable on the typesetter. Each column will have a width 7/15 that of the text line length in single-column format, and a gutter (the space between the columns) of 1/15 of the full line length. Remember that when you use the two-column '.2C' request, either pipe the *nroff* output through *col* or make the first line of the input '.pi /usr/bin/col.'

The '.2C' request is actually a special case of the '.MC' request that produces formats of more than two spaces:

.MC [*column width* [*gutter width*]]

This formats output in as many columns of ***column width*** as will fit across the page with a gap of *gutter width*. You can specify the column width in any unit of scale, but if you do not specify a unit, the setting defaults to ens. '.MC' without any column width is the same thing as '.2C'. To return to single-column output, use '.1C'. For example:

```
.MC
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
```

Switching from double to single-column always causes a skip to a new page.

2.4.7. Footnotes — '.FS' and '.FE'

Material placed between lines with the commands '.FS' (footnote) and '.FE' (footnote end) is collected, remembered, and placed at the bottom of the current page.* The formatting of the footnote is:

```
at the bottom of the current page.*
.FS
* Like this.
.FE
```

By default, footnotes are 11/12th the length of normal text, but you can modify this by changing the FL register (see *Modifying Default Features*). When typeset, footnotes appear in smaller size type.

Because the macros only save a passage of text for printing at the bottom of the page, you have to mark the footnote reference in some way, both in the text preceding the footnote and again as part of the footnote text. We use a simple asterisk, but you can use anything you want.

You can also produce automatically numbered footnotes.‡ Footnote numbers are printed by a pre-defined string (**), which you invoke separately from '.FS' and '.FE'. Each time this string is used, it increases the footnote number by one, whether or not you use '.FS' and '.FE' in your text. Footnote numbers are superscripted on the phototypesetter and on daisy-wheel terminals, but on low-resolution devices (such as the lpr and a crt), they are bracketed. If you use ** to indicate numbered footnotes, the '.FS' macro automatically includes the footnote number at the bottom of the page. This footnote, for example, was produced as follows:[2]

```
This footnote, for example, was produced as follows:\**
.FS
text
.FE
```

If you are using '**' to number footnotes, but want a particular footnote to be marked with an

---

* Like this.

[2] 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, the old -ms macro package.

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 is temporarily suspended, because the '**' string is not used. Instead of a dagger, you could use an asterisk * or double dagger ‡, represented as '\(dd'.

2.4.8. Endnotes

If you want 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 the '.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 gives you numbers at the left-hand margin. With some styles of endnotes, you would want to use '.PP' rather than '.IP' macros, and specify ** before the reference begins.

2.4.9. Displays and Tables — '.DS' and '.DE'

To prepare displays of lines, such as tables, in which the lines should not be re-arranged or broken between pages, enclose them in the requests '.DS' and '.DE':

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

which produces:

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

By default, lines between '.DS' and '.DE' are indented from the left margin.

If you don't want the indentation, use '.DS L' to begin and '.DE' to produce a left-justified display:

to get
something like
this

You can also center lines with the '.DS C' and '.DE' requests:

† In the footnote, the dagger will appear where the footnote number would otherwise appear, as on the left.

This is an
example
of a centered display.

Note that each line is centered individually.

A plain '.DS' is equivalent to '.DS I', which indents and left-adjusts. An extra argument to the '.DS I' or '.DS' request is taken as an amount to indent. For example, '.DS I 3' or '.DS 3' begins a display to be indented 3 ens from the margin.

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', and '.BD' in place of the requests '.DS C', '.DS L', and '.DS B' respectively. Use '.ID' for either a plain '.DS' or '.DS I'. You can also specify the about of indention with the '.ID' macro.

Use the following table as a quick reference:

Table 2-1: Display Macros

Macro with Keep	Macro without Keep
.DS I	.ID
.DS L	.LD
.DS C	.CD
.DS B	.BD
.DS	.ID

Note: It is tempting to assume that '.DS R' will right adjust lines, but it doesn't work.

2.4.10. Keeping Text Together — '.KS', '.KE' and '.KF'

If you wish to keep a table or other block of lines together on a page, there are 'keep - release' requests. 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. There is also a 'keep floating' request. 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 to the top of the next page. *Nroff* fills in the current page with the ordinary text that follows the keep in the input file to avoid leaving blank space at the bottom of the page preceding the keep. Thus, no large blank space will be introduced in the document.

In multi-column output, the keep macros attempt to place all the kept material in the same column.

If the material enclosed in a keep requires more than one page, or more than a column in multi-column format, it will start on a new page or column and simply run over onto the following page or column.

2.4.11. Boxing Words or Lines — '.BX' and '.B2' and '.B2'

To draw rectangular boxes around words, use the request

```
.BX word
```

to print word as shown.

You can box longer pieces of text by enclosing them with '.B1' and '.B2':

```
.B1
Tom appeared on the sidewalk with a bucket of whitewash and a long-handled
brush.
He surveyed the fence, and all gladness left him and a deep melancholy
settled down upon his spirit.
Thirty yards of board fence nine feet high.
Life to him seemed hollow, and
existence but a burden.
.B2
```

This produces:

Tom appeared on the sidewalk with a bucket of whitewash and a long-handled brush. He surveyed the fence, and all gladness left him and a deep melancholy settled down upon his spirit. Thirty yards of board fence nine feet high. Life to him seemed hollow, and existence but a burden.

2.4.12. Changing Fonts — '.I', '.B', '.R' and '.UL'

To get italics on the typesetter or reverse display on the workstation, say:

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

as was done for *these three words*. The '.R' request restores the normal (usually Roman) font. If only one word is to be italicized, you can put it on the line with the '.I' request:

```
.I word
```

and in this case you do not need to use an '.R' to restore the previous font.

You can print **boldface** font by

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

As with '.I', you can place a single word in boldface font by putting it on the same line as the '.B' request. Also, when '.I' or '.B' is used with a word as an argument, it can take as a second argument any trailing punctuation to be printed immediately after the word but set in normal typeface. For example:

```
.B word )
```

prints

word)

that is, the word in boldface and the closing parenthesis in normal Roman directly adjacent to the word.

If you want actual underlining as opposed to italicizing on the typesetter, use the request

.UL word

to underline a word. There is no way to underline multiple words on the typesetter.

2.4.13. Changing the Type Size — '.LG', '.SM' and '.NL'

You can specify a few size changes in *troff* output with the requests '.LG' (make larger), '.SM' (make smaller), and '.NL' (return to normal size). The size change is two points (see *Dimensions* for a discussion of point size); you can repeat the requests for increased effect (here one '.NL' canceled two '.SM' requests). These requests are primarily useful for temporary size changes for a small number of words. They do not affect vertical spacing of lines of text. See *Modifying Default Features* for other techniques for changing the type size and vertical spacing of longer passages.

2.4.14. Dates — '.DA' and '.ND'

When you use *-ms*, *nroff* prints the date at the bottom of each page, but *troff* does not. Both *nroff* and *troff* print it on the cover sheet if you have requested one with '.RP'. To make *troff* print the date as the center page footer, say '.DA' (date). To suppress the date, say '.ND' (no date). To lie about the date, type '.DA July 4, 1776,' which puts the specified date at the bottom of each page. The request:

.ND September 16, 1959

in '.RP' format places the specified date on the cover sheet and nowhere else. Place either '.ND' or '.DA' before the '.RP'. Notice this is one instance that you do not need to put double quote marks around the arguments.

2.4.15. Thesis Format — '.TM'

To format a paper as a thesis, use the '.TM' macro‡ (thesis mode). It is much like the '.th' macro in the *-me* macro package. It puts page numbers in the upper right-hand corner, numbers the first page, suppresses the date, and doublespaces everything except quotes, displays, and keeps. Use it at the top of each file making up your thesis. Calling '.TM' defines the '.CT'‡ macro for chapter titles, which skips to a new page and moves the page number to the center footer. You can use the '.P1' (P one) macro 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.

2.4.16. Bibliography — '.XP'

To format bibliography entries,‡ use the '.XP' macro, which stands for *exdented paragraph.* It exdents the first line of the paragraph by \n(PI units, usually 5n, the same as the indent for the first line of a '.PP'. An example of exdented paragraphs is:

```
.XP
Lumley, Lyle S., \fISex in Crustaceans: Shell Fish Habits,\fP\|
Harbinger Press, Tampa Bay and San Diego, October 1979.
243 pages.
The pioneering work in this field.
.XP
Leffadinger, Harry A., "Mollusk Mating Season: 52 Weeks, or All Year?"
in \fIActa Biologica,\fP\| vol. 42, no. 11, November 1980.
A provocative thesis, but the conclusions are wrong.
```

which produces:

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.

You do have to italicize the book and journal titles and quote the title of the journal article. You can change the indentation and exdentation by setting the value of number register PI.

2.4.17. Table of Contents — '.XS', '.XE', '.XA', '.PX'

There are four macros that produce a table of contents.‡ Enclose table of contents entries 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. A sample of typical input and output text is:

```
.XS ii
Introduction
.XA 1
Chapter 1: Review of the Literature
.XA 23
Chapter 2: Experimental Evidence
.XE
.PX
```

Table of Contents

You can also use the '.XS' and '.XE' pairs 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 make the table of contents after finishing your document. To print out the table of contents, use the '.PX' macro or nothing will happen.

2.4.18. Defining Quotation Marks

To produce quotation marks‡ and dashes that format correctly with both *nroff* and *troff*, there are some string definitions for each of the formatting programs. The *- string yields two hyphens in *nroff*, and produces an em dash — like this one in *troff*. The *Q and *U strings produce " and " in *troff*, but " in *nroff*.

2.4.19. Accent Marks

To simplify typing certain foreign words, the **-ms** package defines strings representing common accent marks. There are 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. Unlike the old accent marks, Place '.AM' (accent mark) at the beginning of your document, and type the accent strings ***after*** the letter being accented.

For the old accent marks, type the string ***before*** the letter over which the mark is to appear. For example, to print 'tél'ephone with the old macros, you type:

```
t\*'el\*'ephone
```

A list of both sets of diacritical marks and examples of what they look like follows. ***Note:*** Do not use the *tbl* macros '.TS' and '.TE' with any of the accent marks as the marks do not line up correctly.

Table 2-2: Old Accent Marks

Accent Name	Input	Output
acute	*'e	é
grave	*`e	è
umlaut	*:u	ü
circumflex	*^e	ê
tilde	*~a	ã
hacek	*Cr	ř
cedille	*,c	ç

Table 2-3: Accent Marks

Accent Name	Input	Output
acute	e*'	é
grave	e*`	è
circumflex	o*^	ô
cedilla	c*,	ç
tilde	n*~	ñ
question	*?	¿
exclamation	*!	¡
umlaut	u*:	ü
digraphes	*8	ß
hacek	c*v	č
macron	a*_	ā
o-slash	o*/	ø
yogh	kni*3t	kniȝt
angstrom	a*o	å
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 of these marks will not print at all, and others will be placed on the wrong letter.

2.5. Modifying Default Features

The **-ms** macro package supplies a standard page layout style. The text line has a default length of six inches; the indentation of the first line of a paragraph is five ens; the page number is printed at the top center of every page after page one; and so on for standard papers. You can alter many of these default features by changing the values that control them.

The computer memory locations where these values are stored are called ***number registers*** and ***string registers***. Number and string registers have names like those of requests, one or two characters long. For instance, the value of the line length is stored in a number register named LL. Unless you give a request to change the value stored in register LL, it will contain the standard or default value assigned to it by ***-ms***. The ***Summary of -ms Number Registers*** table lists the number registers you can change along with their default values.

2.5.1. Dimensions

To change a dimension like the line length from its default value, reset the associated number register with the *nroff* request '.nr' (number register):

```
.nr LL 5i
```

The first argument, 'LL', is the name of a number register, and the second, '5i' is the value being assigned to it. In the case above, the line length is adjusted from the default 6 inches to five inches. As another example, consider:

```
.nr PS 9
```

which makes the default point size 9 point.

The value may be expressed as an integer or may contain a decimal fraction. When setting the value of a number register, it is almost always necessary to include a unit of scale immediately after the value. In the example above, the 'i' as the unit of scale lets *nroff* know you mean five inches and not five of some other unit of distance. But the point size (PS) and vertical spacing (VS) registers are exceptions to this rule; ordinarily they should be assigned a value as a number of points *without indicating the unit of scale.* For example, to set the vertical spacing to 24 points, or one-third of an inch (double-spacing), use the request:

```
.nr VS 24
```

In the unusual case where you want to set the vertical spacing to more than half an inch (more than 36 points), include a unit of scale in setting the VS register. The *Units of Measurement in 'nroff' and 'troff'* table explains the units of measurement.

Table 2-4: Units of Measurement in *nroff* and *troff*

Unit	Abbr	*nroff*	*troff*
point	p	1/72 inch	1/72 inch
pica	p	1/6 inch	1/6 inch
em	m	width of one character	distance equal to number of points in the current typesize
en	n	width of one character	half an em
vertical space	v	amount of space in which each line of text is set, measured baseline to baseline	same
inch	i	inch	inch
centimeter	c	centimeter	centimeter
machine unit	u	1/240 inch	1/432 inch

The units *point*, *pica*, *em*, and *en* are units of measurement used by tradition in typesetting. The *vertical space* unit also corresponds to the typesetting term *leading*, which refers to the distance from the baseline of one line of type to the baseline of the next. Em and en are particulary interesting in that they are proportional to the type size currently in use (normally expressed as a number of points). An em is the distance equal to the number of points in the type size (roughly the width of the letter 'm' in that point size), while an en is half that (about the width of the letter 'n'). These units are convenient for specifying dimensions such as indentation. In *troff*, em and en have their traditional meanings, that is one em of distance is equal to two ens. For *nroff*, on the other hand, em and en both mean the same quantity of distance, the width of one typewritten character.

The *machine unit* is a special unit of dimension used by *nroff* and *troff* internally. This is the unit to which the programs convert almost all dimensions when storing them in memory, and is included here primarily for completeness. In using the features of *-ms*, it is sufficient to know that such a unit of measure exists.

Note that a change to a number register such as LL does not immediately change the related dimension at that point in the output. Instead, in the case of the line length for example, the change takes place at the beginning of the next paragraph, where *-ms* resets various dimensions to the current values of the related number registers.

If you need the effect immediately, use the normal *troff* command in addition to changing the number register. For example, to control the vertical spacing immediately, use:

```
.vs
```

This takes effect at the place where it occurs in your input file. Since it does not change the VS register, however, its effect lasts only until the beginning of the next paragraph. As a general rule, to make a permanent change, or one that will last for several paragraphs until you want to change it again, alter the value of the *-ms* register. If the change must happen immediately, somewhere other than the point shown in the table, use the *nroff* request. If you want the change to be both immediate and lasting, do both.

Table 2-5: Summary of –ms Number Registers

Register	Controls	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. Use the *nroff* '.ds' (define string) request to alter the string registers, as you use the '.nr' request for number registers. The page number on *output* is taken from register PN, to permit changing its output style. For more complicated headers and footers, you can redefine the macros PT and BT, as explained earlier. See *Register Names* for a full list.

2.6. Using 'nroff/troff' Requests

You can use a small subset of the *nroff* requests to supplement the *–ms* macro package.

Use '.nr' and '.ds' requests to manipulate the *–ms* number and string registers as described in *Modifying Default Features*. You can also freely use the other following requests in a file for processing with the *-ms* macro package. They all work with both typesetter and workstation or terminal output.

.ad b — Adjust both margins. This is the default adjust mode.

.bp — Begin new page.

.br — 'Break' line; start a new output line whether or not the current one has been completely filled with text.

.ce *n* — Center the following *n* input text lines individually in the output. If *n* is omitted, only the next line of text is centered.

.ds *XX* — Define string register.

.na — Turn off adjusting of right margins to produce ragged right.

.nr *XX* — Define number register.

.sp n — Insert *n* blank lines. If *n* is omitted, one blank line is produced (the current value of the unit *v*). You can attach a unit of dimension to *n* to specify the quantity in units other than a number of blank lines.

Note: The macro package executes sequences of *nroff* requests on its own, in a manner invisible to you. By inserting your own *nroff* requests, you run the risk of introducing errors. The most

likely result is simply for your *nroff* requests to be ignored, but in some cases the results can include fatal *nroff* errors and garbled typesetter output.

As a simple example, if you try to produce a centered heading with the input:

```
.ce
.SH
Text of section heading
```

you will discover that the heading comes out left-adjusted; the '.SH' macro, appearing after the '.ce' request overrules it and forces left-adjusting. But consider the following sequence:

```
.sp
.ce
.B
Line of text
```

which successfully produces a centered, boldface heading preceded by one line of vertical space. There are lots of tricks like this, so be careful.

To learn more about *troff* see *Formatting Documents with 'nroff' and 'troff'*.

2.7. Using –ms with tbl to Format Tables

Similar to the *eqn* macros are the macros '.TS' and '.TE' defined to separate tables (see *Formatting Tables with 'tbl'*) 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.

2.8. Using –ms with eqn to Typeset Mathematics

If you have to print Greek letters or mathematical equations, see *Typesetting Mathematics with 'eqn'* 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 to '.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).

2.9. Register Names

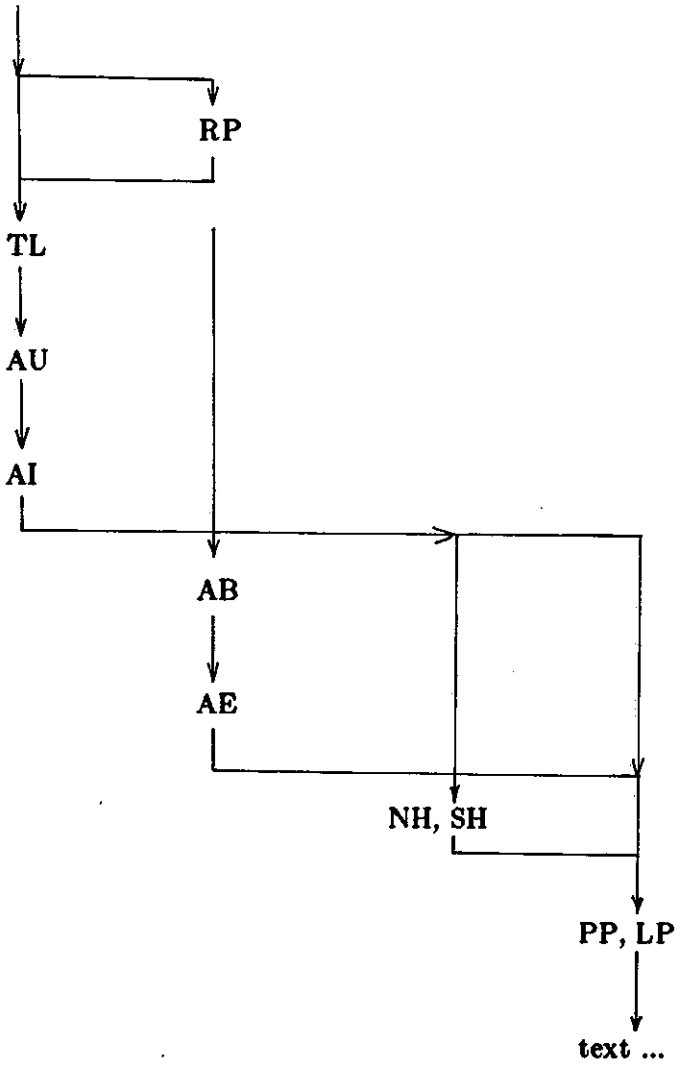
The *-ms* macro package uses the following register names internally. Independent use of these names in your own macros may produce incorrect output. Note that there are no lower-case letters in any *-ms* internal name.

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
T.	FC	H2	IF	IT	MF	ND	PE	PS	TC	WF
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

2.10. Order of Requests in Input

The following diagram provides a quick reference on how to order requests when using the –ms macro package to format a document in released format. For simplier documents, start with an '.LP' initializing request.



2.11. -ms Request Summary

This section includes tables of the old Bell Laboratories that have been removed from the new -ms package, of new -ms requests and string definitions, and of useful printing and displaying commands. It also includes a complete -ms request and string summary for easy reference.

Table 2-6: Old Bell Laboratories Macros

Macro	Explanation
.CS	Cover sheet
.EG	BTL Engineer's Notes
.HO	Bell Labs, Holmdel, N.J.
.IH	Bell Labs, Naperville, Ill.
.IM	BTL internal memo
.MF	BTL file memo
.MH	Bell Labs, Murray Hill, N.J.
.MR	BTL record memo
.ND	BTL date
.OK	BTL keywords for tech memo
.PY	Bell Labs, Piscataway, N.J.
.SG	Signatures for tech memo
.TM	BTL technical memo
.TR	BTL report format.
.WH	Bell Labs, Whippany, N.J.

Table 2-7: New -ms Requests

Request	Explanation
.AM	New accent mark definitions.
.CT	Chapter title in '.TM' format.
.EH	Define even three-part page header.
.EF	Define even three-part page footer.
.FE	End automatically numbered footnote.
.FS	Begin automatically numbered footnote.
.IP **	Number endnotes.
.IX	Index words.
.OF	Define odd three-part page footer.
.OH	Define odd three-part page header.
.P1	Put header on page one in '.TM' format.
.PX	Print table of contents.
.TM	Thesis format.
.XS	Start table of contents entry.
.XE	End table of contents entry.
.XA	Additional table of contents entry.
.PX	Prints table of contents.
.XP	Exdented paragraph.

Table 2-8: New String Definitions

Definition	In nroff	In troff
*-	Two hyphens --	Em dash —
*Q	Open quote "	Open quote "
*U	Closed quote "	Closed quote "

Table 2-9: **-ms** Request Summary

Request	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	–	yes	End abstract.
.AI	no	yes	Author's institution follows.
.AM	–	no	New accent mark definitions
.AT	no	yes	Print '...Attached' and turn off line filling.
.AU	no	yes	Author's name follows.
.B *x*	no	no	Print *x* in boldface; if no argument switch to boldface.
.B1	no	yes	Begin text to be enclosed in a box.
.B2	no	yes	End text to be boxed and print it.
.BT	date	no	Bottom title, automatically invoked at foot of page. May be redefined.
.BX *x*	no	no	Print *x* in a box.
.CM	if t	no	Cut mark between pages.
.CT	if TM	yes	Chapter title in '.TM' only. Page number moved to CF.
.DA *x*	*nroff*	no	'Date line' at bottom of page is *x*. Default is today.
.DE	–	yes	End displayed text. Implies '.KE'.
.DS *x*	no	yes	Start of displayed text to appear verbatim liine-by-line. *x*=I for indented display (default), *x*=L for left-adjusted on the page, *x*=C for centered, *s*=B for make left-justified block, then center whole block. Implies '.KS'.
.EF *x*	–	no	Even three-part page footer *x*
.EN	–	yes	Space after equation produced by *eqn* or *neqn*.

Request	Initial Value	Cause Break	Explanation
.EQ *x y*	–	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be I to indent equation (default), L to left-adjust the equation, or C to center it.
.FE	–	yes	End footnote.
.FS *x*	–	no	Start footnote. *x* is optional footnote label. The note will be printed at the bottom of the page.
.I *x*	no	no	Italicize *x*; if *x* is missing, italic text follows.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	–	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	yes	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.ND *date*	*troff*	no	Use date supplied if any as page footer; only in special format positions.
.NH *n*	–	yes	Same as '.SH' with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.IX *x y*	–	yes	Index entries *w* and *y* and so on up to 5 levels. Make letters normal size.
.OF *x*	–	no	Odd three-part page footer.
.OH *header*	–	no	Odd three-part page header.
.P1	if TM	no	Print header on first page.
.PP	no	yes	Begin paragraph. First line indented.
.PT	pg #	–	Page title, automatically invoked at top of page. May be redefined.
.PX *x*	–	yes	Print table of contents; *x*=no suppresses title.
.QP	–	yes	Begin single paragraph which is indented and shorter.
.R	yes	no	Roman text follows.
.RE	–	yes	End relative indent level.

Request	Initial Value	Cause Break	Explanation
.RP	no	–	Cover sheet and first page for released paper. Must precede other requests.
.RS	–	yes	Start level of relative indentation. Following '.IP's are measured from current indentation.
.SH	–	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TA *z*...	5...	no	Set tabs in ens. Default is 5 10 15 ...
.TE	–	yes	End table.
.TH	–	yes	End heading section of table.
.TL	no	yes	Title follows.
.TM	off	no	Thesis format.
.TS *z*	–	yes	Begin table; if *z* is H, table has repeated heading on subsequent pages.
.UL *z*	–	yes	Underline argument, even in *troff*.
.XA *z y*	–	yes	Another index entry; *z*=page for no for none, *y*=indent.
.XE	–	*yes*	End index entry or series of '.IX' entries.
.XS z y	–	*yes*	Begin index entry; *z*=page for no for none, *y*=indent.
.UL z	–	*yes*	Underline argument, even in *troff*.

Table 2-10: –ms String Definitions

Name	Definition	In nroff	In troff
quote	*Q	"	“
unquote	*U	"	”
dash	*-	—	—
month of year	*(MO	January	January
current date	*(DY	January 19, 1984	January 19, 1984
automatically numbered footnote	**		

Formatting Documents with nroff and troff

Introduction

This chapter† provides a tutorial user's guide and examples, reference section, and a summary and index for *nroff* and *troff*.

nroff and *troff* are text processors for the Sun system that format text for typewriter-like terminals and for a phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. *nroff* and *troff* offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

nroff and *troff* are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. *nroff* can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking *nroff* (or *troff*) at UNIX command level is

logo% **nroff** *options files* (or **troff** *options files*)

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input. If no file names are given, input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
–o*list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N-M* and means pages *N* through *M;* a initial *-N* means from the beginning to page *N;* and a final *N-* means from *N* to the end.
–n*N*	Number first generated page *N*.
–s*N*	Stop every *N* pages. *nroff* will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline.
–m*name*	Prepends the macro file **/usr/lib/tmac/tmac.***name* to the input *files*.
–r*aN*	Register *a* (one-character) is set to *N*.
–i	Read standard input after the input files are exhausted.
–q	Invoke the simultaneous input-output mode of the **rd** request.
–z	Suppress formatted output. Only message output occurs (from 'tm's and diagnostics.

† The material in this chapter is derived from *Nroff/ Troff User's Manual*, Joseph Ossanna.

nroff Only

-h Output tabs used during horizontal spacing to speed output as well as reduce byte count. Device tab settings assumed to be every 8 nominal character widths. Default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

-T*name* Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype , **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

-e Produce equally-spaced words in adjusted lines, using full terminal resolution.

troff Only

-t Direct output to the standard output instead of the phototypesetter.

-a Send a printable (ASCII) approximation of the results to the standard output.

-p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

Each option is invoked as a separate argument; for example,

logo% **nroff -o***4,8-10* **-T***300S* **-m***abc file1 file2*

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with *nroff* and *troff*. These include the equation preprocessors NEQN and EQN[2] (for *nroff* and *troff* respectively), and the table-construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] is available for multiple-column *nroff* output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that *nroff* produces by default. TK[4] is a 37 Teletype simulator postprocessor for printing *nroff* output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for *troff* that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

logo% **tbl** *files* | **eqn** | **troff -t** *options* | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to *troff*s input; and the third indicates the piping of *troff*s output to TCAT.

1. A troff Tutorial

This tutorial† presents the basic uses of *troff* for producing phototypeset documentation. See *Formatting Documents with 'nroff' and 'troff'*, the first section in this chapter for complete details.

troff is a text-formatting program, for producing high-quality printed output from the phototypesetter on the Sun UNIX operating system. This tutorial is an example of **troff** output.

The single most important rule of using **troff** is not to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** (see *Typesetting Mathematics with 'eqn'*) provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** (see *Formatting Tables with 'tbl'*) provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of 'macro packages' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the '-ms' package (see *Formatting Documents with the '-ms' Macro Package*) provides most of the facilities needed for a wide range of document preparation. (This memo was prepared with '-ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX, and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

† The material in this tutorial is derived from *A TROFF Tutorial*, Brian W. Kernighan.

12. Conditionals
13. Environments
14. Diversions

Appendix: Typesetter character set

To use **troff** you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. For **troff** the text and the formatting information are often intertwined quite intimately. Most commands to **troff** are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area $= \pi r^2$

you have to type

```
Area = \(*p\fIr\fR\|\s8\u2\d\s0
```

(which we will explain shortly). The backslash character \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command .ps sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are 1/2 inch. There are 15 point sizes, listed below.

6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36

If the number after .ps is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows .ps, **troff** reverts to the previous size, whatever it was. **troff** begins with point size 10, which is usually fine. This document is in 9 point.

The point size can also be changed in the middle of a line or even a word with the in-line command \s. To produce

The POWER of a SUN

type

```
The \s8POWER\s10 of a \s8SUN\s10
```

As above, \s should be followed by a legal point size, except that \s0 causes the size to revert to its previous value.

Relative size changes are also legal and useful:

```
\s-2SUN\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is .vs. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, .ps and .vs revert to the previous size and vertical spacing respectively.

The command .sp is used to get extra vertical space. Unadorned, it gives you one extra blank line (one .vs, whatever that has been set to). Typically, that's more or less than you want, so .sp can be followed by information about how much space you want —

```
.sp 2i
```

means 'two inches of vertical space'.

```
.sp 2p
```

means 'two points of vertical space'; and

```
.sp 2
```

means 'two vertical spaces' — two of whatever .vs is set to (this can also be made explicit with .sp 2v); **troff** also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use .ft R; to return to the previous font, whatever it was, use either .ft P or just .ft. The 'underline' command

```
.ul
```

causes the next input line to print in italics. .ul can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the in-line command \f:

bold*face* text

is produced by

```
\fBbold\fIface\fR text
```

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra \fP commands, like this:

```
\fBbold\fP\fIface\fP\fR text\fP
```

Because only the immediately previous font is remembered, you have to restore the

previous font after each change or you can lose it. The same is true of .ps and .vs when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells **troff** what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate .fp commands should appear at the beginning of your document if you do not use the standard fonts.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, \f3 and .ft 3 mean 'whatever font is mounted at position 3', and thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in the *nroff/troff* reference manual.

Special characters have four-character names beginning with \(, and they may be inserted anywhere. For example,

$1/4 + 1/2 = 3/4$

is produced by

```
\(14 + \(12 = \(34
```

In particular, greek letters are all of the form \(*–, where – is an upper or lower case roman letter reminiscent of the greek. Thus to get

$\Sigma(\alpha \times \beta) \rightarrow \infty$

in bare **troff** we have to type

```
\(*S(\(*a\(mu\(*b) \(-> \(if
```

That line is unscrambled as follows:

\(*S	Σ
(	(
\(*a	α
\(mu	$\times$
\(*b	β
)	)
\(->	$\rightarrow$
\(if	∞

A complete list of these special names occurs in Appendix A.

In **eqn** (*Formatting Mathematics with 'eqn'*) the same effect can be achieved with the input

```
SIGMA ( alpha times beta ) -> inf
```

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned — the 'translate' command

```
.tr \(mi\(em
```

is perfectly clear, meaning

```
.tr —-
```

that is, to translate – into —.

Some characters are automatically translated into others: grave ` and acute ' accents (apostrophes) become open and close single quotes '~'; the combination of "..." is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit – sign, use \-. To get a backslash printed, use \e.

4. Indents and Line Lengths

troff starts with a line length of 6.5 inches, too wide for 8-1/2×11 paper. To reset the line length, use the .ll command, as in

```
.ll 6i
```

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by the .po command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command .in causes the left margin to be indented by some specified amount from the page offset. If we use .in to move the left margin in, and .ll to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll -0.3i
text to be set into a block
.ll +0.3i
.in -0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of '+' and '–' to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch *long*.

With .in, .ll and .po, the previous value is used if no argument is specified.

To indent a single line, use the 'temporary indent' command .ti. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for .ti, as for most horizontally oriented commands (.ll, .in, .po), is ems; an em is roughly the width of the letter 'm' in the current point size.

Thus, an em is always proportional to the point size you are using. An em in size *p* is the number of *p* points in the width of an 'm'. Here's an em followed by an em dash in several point sizes to show why this is a *proportional* unit of measure. You wouldn't want a 20-point dash if you are printing the rest of a document in 12-point. Here's 12-point:

m
|—|

Here's 16-point:

m
|—|

And here's 20-point:

m
|—|

Thus a temporary indent of .ti 3 in the current point size results in an indent of three m's width or |mmm|.

Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in .ti 2.5m.

Lines can also be indented negatively if the indent is already positive:

.ti -0.3i

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter 'P' back with a .ti command:

Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the 'P' bigger (just a '\s36\s0'), and to move it down from its normal position (see the section on local motions).

5. Tabs

Tabs (the ASCII 'horizontal tab' character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the .ta command. To set stops every inch, for example,

.ta 1i 2i 3i 4i 5i 6i

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, ***don't*** use **troff** directly; use the **tbl** program described in ***Formatting Tables with 'tbl'***.

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

```
  1          2          3
 40         50         60
700        800        900
```

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the .tc command:

```
.ta 1.5i 2.5i
.tc \(ru    (\(ru is "_")
Name tab Age tab
```

produces

Name _____ Age _____

To reset the tab replacement character to a blank, use .tc with no argument. (Lines can also be drawn with the \l command, described in Section 6.)

troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by **tbl**). We will not go into it in this tutorial.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2' and the big 'P' in the Paternoster. How are they done? **troff** provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use **eqn**, subscripts and superscripts are most easily done with the half-line local motions \u and \d. To go back up the page half a point-size, insert a \u at the desired place; to go down, insert a \d. (\u and \d should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with \s-2...\s0. Since \u and \d refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by \u and \d isn't the right amount. The \v command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v'(amount)'
```

causes motion up or down the page by the amount specified in '(amount)'. For example,

to move the 'P' down, we used

```
.in + 0.6i      (move paragraph in)
.ll -0.3i       (shorten lines)
.ti -0.3i       (move P back)
\v'1'\s36P\s0\v'-1'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus \v'-1' causes an upward vertical motion of one line space.

There are many other ways to specify the amount of motion —

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

and so on are all legal. Notice that the scale specifier i or p or m goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other **troff** commands described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus \v, like \u and \d, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available — \h is quite analogous to \v, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so **eqn** replaces this by

```
>\h'-0.3m'>
```

to produce >>.

Frequently \h is used with the 'width function' \w to generate motions equal to the width of some character string. The construction

```
\w'thing'
```

is a number equal to the width of 'thing' in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an 'x', we can say

```
\h'\w'x'u'
```

As we mentioned above, the default scale factor for all horizontal dimensions is m, ems, so here we must have the u for machine units, or the motion produced will be far too large. **troff** is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like .sp, were done by overstriking with a slight offset. The commands for .sp are

```
.sp\h'-\w'.sp'u'\h'1u'.sp
```

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp'

again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen \0, which is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also \(blank), which is an unpaddable character the width of a space, \|, which is half that width, \^, which is one quarter of the width of a space, and \&, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command \o, used like

\o'set of characters'

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which makes

système téléphonique

The accents are \(ga and \(aa, or \` and \'; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, \z, the zero-motion command. \zx suppresses the normal horizontal motion after printing the single character x, so another character can be laid on top of it. Although sizes can be changed within \o, it centers the characters on the widest, and there can be no horizontal or vertical motions, so \z may be the only way to get what you want:

is produced by

```
.sp 2
\s8\z\(ci\s14\z\(ci\s22\z\(ci\s36\z\(ci
```

The .sp is needed to leave room for the result.

As another example, an extra-heavy semicolon that looks like

; instead of ; or ;

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v'-0.25m'.\v'0.25m'\s0
```

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function \b, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

$$\left\{\left[x\right]\right\}$$

by typing in only this:

```
.sp
\b'\(lt\(lk\(lb' \b'\(lc\(lf' x \b'\(rc\(rf' \b'\(rt\(rk\(rb'
```

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. **\l'1i'** draws a line one inch long, like this: _____. The length can be followed by the character to use if the _ isn't appropriate; **\l'0.5i.'** draws a half-inch line of dots: ..... The construction **\L** is entirely analogous, except that it draws a vertical line instead of horizontal.

7. Strings

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing **\o"e\'"** for each é would be a great nuisance.

Fortunately, **troff** provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command .ds. The line

```
.ds e \o"e\'"
```

defines the string e to have the value **\o"e\'"**

String names may be either one or two characters long, and are referred to by ***x** for one character names or ***(xy** for two character names. Thus to get téléphone, given the definition of the string e as above, we can say t*el*ephone.

If a string must begin with blanks, define it as

```
.ds xx "      text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a \ at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in **troff**, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a

string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a **troff** 'command' like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro*. The way we tell **troff** what .PP means is to *define* it with the .de command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used '.PP' for 'paragraph', and upper case so it wouldn't conflict with any name that **troff** might already know about). The last line .. marks the end of the definition. In between is the text, which is simply inserted whenever **troff** sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of .PP has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of .PP to something like

```
.de PP      \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used .PP.

\" is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS      \" start indented block
.sp
.nf
.in + 0.3i
..
.de BE      \" end indented block
.sp
.fi
.in -0.3i
..
```

Now we can surround text like

> Copy to
> John Doe
> Richard Roberts
> Stanley Smith

by the commands .BS and .BE, and it will come out as it did above. Notice that we indented by .in + 0.3i instead of .in 0.3i. This way we can nest our uses of .BS and BE to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of .BS and .BE, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

~~~~left top center top right top~~~~

In **roff**, one can say

```
.he 'left top'center top'right top'
.fo 'left bottom'center bottom'right bottom'
```

to get headers and footers automatically on every page. Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro .NP (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command 'bp, which causes a skip to top-of-page (we'll explain the ' shortly). Then we space down half an inch, print the title (the use of .tl should be self explanatory; later we will discuss
~~~~

parameterizing the titles), space another 0.3 inches, and we're done.

To ask for .NP at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command .wh:

```
.wh -1i NP
```

(No '.' is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so '-1i' means 'one inch from the bottom'.

The .wh command appears in the input outside the definition of .NP; typically the input would be

```
.de NP
...
..
.wh -1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the .NP macro is activated. (In the jargon, the **.wh** command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) .NP causes a skip to the top of the next page (that's what the **'bp** was for), then prints the title with the appropriate margins.

Why **'bp** and **'sp** instead of .bp and .sp? The answer is that .sp and .bp, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used .sp or .bp in the .NP macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using **'** instead of . for a command tells **troff** that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp  .br  .ce  .fi  .nf  .sp  .in  .ti
```

All others cause *no* break, regardless of whether you use a . or a **'**. If you really need a break, add a .br command at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the .lt command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change .NP to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
'bp
'sp 0.5i
.ft R        \" set title font to roman
.ps 10       \" and size to 10 point
.lt 6i       \" and length to 6 inches
.tl 'left'center'right'
.ps          \" revert to previous size
.ft P        \" and to previous font
'sp 0.3i
..
```

This version of .NP does *not* work if the fields in the .tl command contain size or font changes. To cope with that requires **troff**'s 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify .NP so it does some processing before the 'bp command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character % in the .tl line at the position where you want the number to appear. For example

```
.tl "- % -"
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either .bp n, which immediately starts a new page numbered n, or with .pn n, which sets the page number for the next page but doesn't cause a skip to the new page. Again, .bp + n sets the page number to n more than its current value; .bp means .bp + 1.

10. Number Registers and Arithmetic

troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the .nr command, and are referenced anywhere by \nx (one character name) or \n(xy (two character name).

There are quite a few pre-defined number registers maintained by **troff**, among them % for the current page number; nl for the current vertical position on the page; dy, mo and yr for the current day, month and year; and .s and .f for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like .s and .f, cannot be changed with .nr.

As an example of the use of number registers, in the -ms macro package (*Formatting Documents with the '-ms' Macro Package*). most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and

vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro .PP is defined (roughly) as follows:

```
.de PP
.ps \\n(PS      \" reset size
.vs \\n(VSp     \" spacing
.ft R           \" font
.sp 0.5v        \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers PS and VS.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When **troff** originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used*, we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes is only needed for \n, *, \$ (which we haven't come to yet), and \ itself. Things like \s, \f, \h, \v, and so on do not need an extra backslash, since they are converted by **troff** to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS-2
```

decrements PS by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, =, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. **troff** arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus

```
7*-4+3/13
```

becomes '-1'. Number registers can occur anywhere in an expression, and so can scale indicators like p, i, m, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so 1i/2u evaluates to 0.5i correctly.

The scale indicator u often has to appear when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3 inches. Sorry. Remember that the default units for

horizontal parameters like .ll are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a .nr command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the u on the .ll command.

11. Macros with arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro .SM that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of .SM is

```
.de SM
\s-2\\$1\s+2
..
```

Within a macro definition, the symbol \\$n refers to the nth argument that the macro was called with. Thus \\$1 is the string to be placed in a smaller point size when .SM is called.

As a slightly more complicated version, the following definition of .SM permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

produces (TROFF). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

By the way, the number of arguments that a macro was called with is available in number register .$.

The following macro .BD is the one used to make the 'bold roman' we have been using for troff command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h'-\w'\\$1'u+1u'\\$1\fP\\$2
..
```

The \h and \w commands need no extra backslash, as we discussed above. The \& is there in case the argument begins with a period.

Two backslashes are needed with the \\$n commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called .SH which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the .SH macro:

```
.nr SH 0    \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1\" increment number
.ps \\n(PS-1    \" decrease PS
\\n(SH. \\$1    \" number. title
.ps \\n(PS      \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used \\n(SH instead of \n(SH and \\n(PS instead of \n(PS. If we had used \n(SH, we would get the value of the register at the time the macro was *defined*, not at the time it was *used*. If that's what you want, fine, but not here. Similarly, by using \\n(PS, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our .NP macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl '\\*(LT'\\*(CT'\\*(RT'
```

so the title comes from three strings called LT, CT and RT. If these are empty, then

the title will be a blank line. Normally CT would be set with something like

```
.ds CT - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .SH macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the .SH macro whether the section number is 1, and add some space if it is. The .if command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i    \" first section only
```

The condition after the .if can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro .S1 and invoke it if we are about to do section 1 (as determined by an .if).

```
.de S1
--- processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown or you will get unexpected extra lines in your output. troff also provides an 'if-else' construction, which we will not go into here.

A condition can be negated by preceding it with !; we get the same effect as above (but less clearly) by using

```
.if !\\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with .if. For example, is the current page even or odd?

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are t and n, which tell you whether the formatter is troff or nroff.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an .if:

```
.if 'string1'string2' stuff
```

does 'stuff' if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with *, arguments with \$, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **troff** provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command .ev n shifts to environment n; n must be 0, 1 or 2. The command .ev with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro .NP to process titles in environment 1 like this:

```
.de NP
.ev 1       \" shift to new environment
.lt 6i      \" set parameters here
.ft R
.ps 10
... any other processing ...
.ev         \" return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the .NP macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command .di xy begins a diversion — all subsequent output is collected into the macro xy until the command .di with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register dn.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands .KS and .KE will not be split across a page boundary (as for a figure or table). Clearly, when a .KS is encountered, we have to begin diverting the output so we can find out how big it is. Then when a .KE is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS  \" start keep
.br     \" start fresh line
.ev 1   \" collect in new environment
.fi     \" make it filled text
.di XX  \" collect in XX
..

.de KE  \" end keep
.br     \" get last partial line
.di     \" end diversion
.if \\n(dn>=\\n(.t .bp  \" bp if doesn't fit
.nf     \" bring it back in no-fill
.XX     \" text
.ev     \" return to normal environment
..
```

Recall that number register nl is the current position on the output page. Since output was being diverted, this remains at its value when the diversion started. dn is the amount of text in the diversion; .t (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the .if is satisfied, and a .bp is issued. In either case, the diverted output is then brought back with .XX. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Appendix A: Phototypesetter Character Set

These characters exist in roman, italic, and bold. To get the one on the left, type the four-character name on the right.

ff	\(ff	fi	\(fi	fl	\(fl	ffi	\(Fi	ffl	\(Fl
_	\(ru	—	\(em	¼	\(14		\(12		\(34
©	\(co	°	\(de	†	\(dg	'	\(fm		\(ct
	\(rg	•	\(bu		\(sq	-	\(hy		

(In bold, \(sq is .)

The following are special-font characters:

+	\(pl	−	\(mi	×	\(mu	÷	\(di
=	\(eq	≡	\(==	≥	\(>=	≤	\(<=
≠	\(!=	±	\(+-	¬	\(no	/	\(sl
~	\(ap	≃	\(~=	≈	\(pt	∇	\(gr
→	\(->	←	\(<-	↑	\(ua	↓	\(da
∫	\(is	∂	\(pd	∞	\(if	√	\(sr
⊂	\(sb	⊃	\(sp	∪	\(cu	∩	\(ca
⊆	\(ib	⊇	\(ip	∈	\(mo	∉	\(es
´	\(aa	`	\(ga	○	\(ci	∅	\(bs
§	\(sc	‡	\(dd	←	\(lh	→	\(rh
⌠	\(lt	⌡	\(rt	⌈	\(lc	⌉	\(rc
⎛	\(lb	⎞	\(rb	⌊	\(lf	⌋	\(rf
⎨	\(lk	⎬	\(rk	⎜	\(bv	φ	\(ts
⎪	\(br	⎪	\(or	_	\(ul	‾	\(rn
*	\(**						

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ` is the other quote mark.

´	\'	`	\`	-	\-	_	_

These characters exist only on the special font, but they do not have four-character names:

" { } < > ~ ^ \ # 0

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(*a is α.

a b g d e z y h i k l m n c o p r s t u f x q w
$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$

A B G D E Z Y H I K L M N C O P R S T U F X Q W
$A\ B\ \Gamma\ \Delta\ E\ Z\ H\ \Theta\ I\ K\ \Lambda\ M\ N\ \Xi\ O\ \Pi\ P\ \Sigma\ T\ T\ \Phi\ X\ \Psi\ \Omega$

Request Reference

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ' (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ' suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function \n*R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in \n*x*, or left-parenthesis-introduced, two-character name as in \n(*xx*.

1.2. Formatter and device resolution. *troff* internally uses 432 units/inch, corresponding to the phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. *nroff* internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. *troff* rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. *nroff* similarly rounds numerical input to the actual resolution of the output device indicated by the **-T** option (default Model 37 Teletype).

1.3. Numerical parameter input. Both *nroff* and *troff* accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units *troff*	*nroff*
i	Inch	432	240
c	Centimeter	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6×*S*	*C*
n	En = Em/2	3×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In *nroff, both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in *nroff* need not be all the same and constructed characters such as -> (→) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling. The number, *N*,

may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

```
.sp |3.2c
```

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, –, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or – is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

```
.ll (4.25i+\nxP+3)/2u
```

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or –*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The *troff* character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(*xx* where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by *troff*	
Character	Name	Character	Name
´	acute accent	'	close quote
`	grave accent	‘	open quote
–	minus	-	hyphen

The characters ´, `, and – may be input by \´, \`, and \– respectively or by their names (Table II). The ASCII characters @, #, ", ', `, <, >, \, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

nroff understands the entire *troff* character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The characters ´, `,

and _ print as themselves.

2.2. Fonts. The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either **\f***x*, **\f(***xx*, or **\f***N* where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not* necessary to change to the Special font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored*. *troff* can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

nroff understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a **\s***N* at the desired point to set the size to *N*, or a **\s±** *N* ($1 \le N \le 9$) to increment/decrement the size by *N*; **\s0** restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. *nroff* ignores type size control.

Request Form	*Initial Value*	*If No Argument*	*Notes**	*Explanation*
.ps ± *N*	10 point	previous	E	Point size set to ± *N*. Alternatively imbed **\s***N* or **\s±** *N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence + *N*, –*N* will work because the previous requested value is also remembered. Ignored in *nroff*.
.ss *N*	12/36 em	ignored	E	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in *nroff*.
.cs *F N M*	off	-	P	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M*-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be 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.
.fs *S F N*	none	-	-	Forces font *F* to be in size *N*. A **.fs 3 –2** causes implicit \s–2 every time font 3 is entered, and a matching \s+2 when left. Same for Special font characters that are used during *F*. Use *S* to handle Special characters during *F* (**fs 3 –3** or **fs S 3 –0** cause automatic reduction of font 3 by 3 points while special characters are not affected. Any **fp** request specifying a font on some position must precede **fs** requests relating to that position.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and –*N* (*N* from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on *nroff* output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl ± *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 effect 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

*The use of "'" as control character (instead of ".") suppresses the break function.

†Values separated by ";" are for *nroff* and *troff* respectively.

				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 output and spring the trap. In a diversion, *D* is the distance to the *diversion trap*, if any, or is very large.
.mk *R*	none	internal	D	Mark the *current* vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See **rt** request.
.rt ± *N*	none	internal	D,v	Return *upward only* to a marked vertical place in the current diversion. If ± *N* (w.r.t. current place) is given, the place is ± *N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous **mk**. Note that the **sp** request (§5.3) may be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; e. g. using the sequence **.mk** *R* ... **.sp |\nR**u.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in *troff* and the minimum interword spacing can be controlled with the **ss** request (§2). In *nroff*, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **–e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with ., ?, or ! is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a **\p** may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character **\&**. Still another way is to specify output translation of some convenient character into the control character using **tr** (§10.5).

4.2. Interrupted text. The copying of an input line in *nofill* (non-fill) mode can be *interrupted* by terminating the partial line with a **\c**. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with **\c**; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.br	-	-	B	**Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.**
.fi	fill on	-	B,E	**Fill subsequent output lines. The register .u is 1 in fill mode and 0 in nofill mode.**
.nf	fill on	-	B,E	**Nofill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.**
.ad *c*	adj,both	adjust	E	**Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.**

Indicator	Adjust Type
l	**adjust left margin only**
r	**adjust right margin only**
c	**center**
b or n	**adjust both margins**
absent	**unchanged**

***c* can be number obtained from j register.**

.na	adjust	-	E	**Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for ad is not changed. Output line filling still occurs if fill mode is on.**
.ce *N*	off	*N*=1	B,E	**Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.**

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing *(V)* between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in *troff*, and to the output device resolution in *nroff*. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; *troff* default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e. g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function **\x'***N*'** can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here '), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.vs *N*	1/6in;12pts	previous	E,p	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x'***N***'** (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set to ± *N*. *N*–1 *Vs (blank lines)* are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.
.sp *N*	-	*N*=1*V*	B,v	Space vertically in *either* direction. If *N* is negative, the motion is *backward* (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see **ns**, and **rs** below).
.sv *N*	-	*N*=1*V*	v	Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has *no* effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see **os**). Subsequent **sv** requests will overwrite any still remembered *N*.
.os	-	-	-	Output saved vertical space. No-space mode has *no* effect. Used to finally output a block of vertical space requested by an earlier **sv** request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits **sp** requests and **bp** requests *without* a next page number. The no-space mode is turned off when a line of output occurs, or with **rs**.
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	B	Causes a break and output of a blank line exactly like **sp 1**.

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ll ± *N*	6.5in	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 ($1 \leq N \leq 9$). 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 an input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra ****) to delay interpolation until argument

reference *time*.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers **dn** and **dl** respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (**cs**) or emboldened (**bd**) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see **it** below.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.de *xx yy*	-	*.yy=..*	-	Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with *.yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with "..". A macro may contain **de** requests provided the terminating macros differ or the contained definition terminator is concealed. ".." can be concealed as \\.. which will copy as \.. and be reread as "..".
.am *xx yy*	-	*.yy=..*	-	Append to macro (append version of **de**).
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*. Any initial double-quote in *string* is stripped off to permit initial blanks.
.as *xx string*	-	ignored	-	Append *string* to string *xx* (append version of **ds**).
.rm *xx*	-	ignored	-	Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.

.dl *zz*	-	end	D	Divert output to macro *zz*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request **dl** or **da** is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.
.da *zz*	-	end	D	Divert, appending to *zz* (append version of **dl**).
.wh *N zz*	-	-	v	Install a trap to invoke *zz* 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 *zz*. A zero *N* refers to the *top* of a page. In the absence of *zz*, the first found trap at *N*, if any, is removed.
.ch *zz N*	-	-	v	Change the trap position for macro *zz* to be *N*. In the absence of *N*, the trap, if any, is removed.
.dt *N zz*	-	off	D,v	Install a diversion trap at position *N* in the *current* diversion to invoke macro *zz*. Another **dt** will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it *N zz*	-	off	E	Set an input-line-count trap to invoke the macro *zz* 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 *zz*	none	none	-	The macro *zz* will be invoked when all input has ended. The effect is the same as if the contents of *zz* 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 **^**, **#^*zzz*^*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* ...	0.8; 0.5in	none	E,m	Set tab stops and types. *t*=**R**, right adjusting; *t*=**C**, centering; *t* absent, left adjusting. *troff* tab stops are preset every 0.5in.; *nroff* every 0.8in. The stop values are separated by spaces, and a value preceded by **+** is treated as an increment to the previous stop value.
.tc *c*	none	none	E	The tab repetition character becomes *c*, or is removed specifying motion.
.lc *c*	.	none	E	The leader repetition character becomes *c*, or is removed specifying motion.
.fc *a b*	off	off	-	The field delimiter is set to *a*; the padding indicator is set to the *space* character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with **tr** (§10.5). *All* others are ignored.

The *escape* character **** introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. **** should not be confused with the ASCII control character ESC of the same name. The escape character **** can be input with the sequence ****. The escape character can be changed with **ec**, and all that has been said about the default **** becomes true for the new escape character. **\e** can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	****	****	-	Set escape character to ****, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current *troff* character set — **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input (even in *nroff*) by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in *troff*, and *automatically* invokes ligatures during input.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	on	-	Ligature mode is turned on if *N* is absent or non-zero, and turned off if *N*=0. If *N*=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in *copy mode*. No effect in *nroff*.

10.3. Backspacing, underlining, overstriking, etc. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

nroff automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and **\f***F*, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	Underline in *nroff* (italicize in *troff*) the next *N* input text lines. Actually, switch to *underline* font, saving the current font for later restoration; *other* font changes within the span of a **ul** will take effect, but the restoration will undo the last change. Output generated by **tl** (§14) *is* affected by the font change, but does *not* decrement *N*. If *N*>1, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu *N*	off	*N*=1	E	A variant of **ul** that causes *every* character to be underlined in *nroff*. Identical to **ul** in *troff*.
.uf *F*	Italic	Italic	-	Underline font set to *F*. In *nroff*, *F* may *not* be on position 1 (initially Times Roman).

10.4. Control characters. Both the control character . and the *no-break* control character ' may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	.	.	E	The basic control character is set to *c*, or reset to ".".
.c2 *c*	'	'	E	The *nobreak* control character is set to *c*, or reset to "'".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tr *abcd....*	none	-	O	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a **\!** is read in *copy mode* and *transparently* output (without the initial **\!**); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape ****. The sequence ****(newline) is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with **\"**. The newline at the end of a comment cannot be concealed. A line beginning with **\"** will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with **.\"**.

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions **\v'***N***'** and **\h'***N***'** can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	*Effect in troff*	*Effect in nroff*	*Horizontal Local Motion*	*Effect in troff*	*Effect in nroff*
\v'*N*'	Move distance *N*		\h'*N*' \(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* 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 1/2em above the current baseline (½ line in *nroff*). For example, **\b'\(lc\(if 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

```
.ul "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and words in a box.

The function **\L'***Nc***'** will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in *nroff*), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (**\(br**); the other suitable character is the *bold vertical* | (**\(bv**). The line is begun without any initial motion relative to the current base line. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn *no* compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the -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

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** in a macro is felt when **so** is encountered. 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 Requests

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.me *c N*	-	off	E,m	Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by **tl**). If the output line is too-long (as can happen in nofill mode) the character will be appended to the line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in *nroff* and 1 em in *troff*.

				The *margin character used with this paragraph was a 12-point box-rule.*
.tm *string*	-	newline	-	After skipping initial blanks, *string* (rest of the line) is read in *copy mode* and written on the user's terminal.
.ig *yy*	-	*.yy=..*	-	Ignore input lines. **ig** behaves exactly like **de** (§7) except that the input is discarded. The input is read in *copy mode*, and any auto-incremented registers will be affected.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in *blocks* of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.
ab *text*	none	-	-	Displays *text* and terminates without further processing. If *text* is missing, 'User Abort' is displayed. Does not cause a break. The output buffer is flushed.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where *nroff* formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various *error* conditions may occur during the operation of *nroff* and *troff*. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in *nroff* and a ⇐ in *troff*. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

Tutorial Examples

T1. Introduction

Although *nroff* and *troff* have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into *nroff* and *troff*. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether *troff* or *nroff* is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd          \"define header
'sp 1i
..              \"end definition
.de fo          \"define footer
'bp
..              \"end definition
.wh 0 hd
.wh -1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ' to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd          \"header
.if t .tl ''\(rn''\(rn'' \"troff cut mark
.if \\n%>1 \{\
'sp |0.5i-1     \"tl base at 0.5i
.tl ''- % -''   \"centered page number
.ps             \"restore size
.ft             \"restore font
.vs \}          \"restore vs
'sp |1.0i       \"space to 1.0i
.ns             \"turn on no-space mode
..
.de fo          \"footer
.ps 10          \"set footer/header size
.ft R           \"set font
.vs 12p         \"set base-line spacing
.if \\n%=1 \{\
'sp |\\n(.pu-0.5i-1  \"tl base 0.5i up
.tl ''- % -''\}  \"first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If *troff* is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the base-line spacing. The *no-space*

*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).

mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s   \"current size
.ps
.nr s2 \\n(.s   \"previous size
. ---           \"rest of footer
..
.de hd
. ---           \"header stuff
.ps \\n(s2      \"restore previous size
.ps \\n(s1      \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn          \"bottom number
.tl "- % -"     \"centered page number
..
.wh -0.5i-1v bn \"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, 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
.fl
\\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
.ln 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 0.2inch, and cannot exceed a length of 0.3inch without intruding into the paragraph. The label could be right adjusted against 0.4inch 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 .fs  \"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
.fl             \"fill mode
..
.de ef          \"end footnote
.br             \"finish output
.nr s \\n(.v    \"save spacing
.ev             \"pop ev
.di             \"end diversion
.nr y -\\n(dn   \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\ns) \
                \"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'11'          \"1 inch rule
.br
..
.de fs          \"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, **fs** 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 **s**. **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 ¼em space. They are Times Roman, Italic, Bold, and a special mathematical font.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• — - _ ¼ fi fl ff ffi ffl ° † ′ ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |*
• — - _ *fi fl ff ffi ffl* ° † ′ © ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• — - _ **fi fl ff ffi ffl** ° † ′ © ©

Special Mathematical Font

” ´ \ ˆ _ ` ˜ / < > { } # @ + − = *
$\alpha\ \beta\ \gamma\ \delta\ \epsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \varphi\ \tau\ \upsilon\ \phi\ \chi\ \psi\ \omega$
$\Gamma\ \Delta\ \Theta\ \Lambda\ \Xi\ \Pi\ \Sigma\ \Upsilon\ \Phi\ \Psi\ \Omega$
$\sqrt{\ }\ \geq\ \leq\ \equiv\ \sim\ \simeq\ \neq\ \rightarrow\ \leftarrow\ \uparrow\ \downarrow\ \times\ \div\ \pm\ \cup\ \cap\ \subset\ \supset\ \subseteq\ \supseteq\ \infty\ \partial$
$\S\ \nabla\ \neg\ \int\ \approx\ \emptyset\ \in\ \ddagger\ \Rightarrow\ \Leftarrow\ \oslash\ |\ \bigcirc\ \backslash\ '\ \{\ \}\ |\ \bigcup\ \bigcap |$

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
	\(12	1/2
	\(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
o	\(*o	omicron
π	\(*p	pi
ρ	\(*r	rho
σ	\(*s	sigma
φ	\(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
∅	\(es	empty set
∈	\(mo	member of
|	\(br	box vertical rule
‡	\(dd	double dagger

Char	*Input Name*	*Character Name*
⇒	\(rh	right hand
⇐	\(lh	left hand
⊘	\(bs	Bell System logo
|	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
⎪	\(bv	bold vertical
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

Summary and Index for Request Reference

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
1. General Explanation				
2. Font and Character Size Control				
.ps ± *N*	10 point	previous	E	Point size; also **\s**± *N*.†
.ss *N*	12/36 em	ignored	E	Space-character size set to *N*/36 em.†
.cs *F N M*	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* = *z*, *zz*, or 1-4. Also **\f***z*, **\f(***zz*, **\f***N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position 1≤*N*≤4.
.fs *S F N*	none	-	-	Forces font *F* or *S* for special characters to be in size *N*.
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* from **j**.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
5. Vertical Spacing				
.vs *N*	1/6in;12pts	previous	E,p	Vertical base line spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*–1 *V*s after each text output line.
.sp *N*	-	*N*=1*V*	B,v	Space vertical distance *N in either direction.*
.sv *N*	-	*N*=1*V*	v	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll ± *N*	6.5 in	previous	E,m	Line length.
.in ± *N*	*N*=0	previous	B,E,m	Indent.
.ti ± *N*	-	ignored	B,E,m	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de *xx yy*	-	*.yy*=..	-	Define or redefine macro *xx;* end at call of *yy*.
.am *xx yy*	-	*.yy*=..	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.

•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*
.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*		-	-	Interpolate source file *name* contents when **so** encountered.
.nx *filename*		end-of-file	-	Next file.
.pi *program*		-	-	Pipe output to *program* (*nroff* only).
20. Miscellaneous Requests				
.ab *text*	none	User Abort	-	Displays *text* and terminates without further processing; output buffer is flushed.
.mc *c N*	-	off	E,m	Set margin character *c* and separation *N*.
.tm *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.ig *yy*	-	*.yy*=..	-	Ignore till call of *yy*.
.pm *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.fl	-	-	B	Flush output buffer.
21. Output and Error Messages				

Notes-

- B Request normally causes a break.
- D Mode or relevant parameters associated with current diversion level.
- E Relevant parameters are a part of the current environment.
- O Must stay in effect until logical output.
- P Mode must be still or again in effect at the time of physical output.

v,p,m,u Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

ad	4	cc	10	ds	7	fc	9	ie	16	ll	6	nh	13	pi	19	rm	7	ta	9	vs	5
af	8	ce	4	dt	7	fi	4	if	16	ls	5	nm	15	pl	3	rr	8	tc	9	wh	7
am	7	ch	7	ec	10	fl	20	ig	20	lt	14	na	15	pm	20	rs	5	ti	6		
as	7	cs	2	el	16	fp	2	in	6	mc	20	nr	8	pn	3	rt	3	tl	14		
bd	2	cu	10	em	7	ft	2	it	7	mk	3	ns	5	po	3	so	19	tm	20		
bp	3	da	7	eo	10	hc	13	lc	9	na	4	nx	19	ps	2	sp	5	tr	10		
br	4	de	7	ev	17	hw	13	lg	10	ne	3	os	5	rd	18	ss	2	uf	10		
c2	10	di	7	ex	18	hy	13	li	10	nf	4	pc	14	rm	7	sv	5	ul	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	\(*zz*	Character named *zz*
7.1	**z*, *(*zz*	Interpolate string *z* or *zz*
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*z*,\f(*zz*,\f*N*	Change to font named *z* or *zz*, or position *N*
11.1	\h'*N*'	Local horizontal motion; move right *N (negative left)*
11.3	\k*z*	Mark horizontal *input* place in register *z*
12.4	\l'*Nc*'	Horizontal line drawing function (optionally with *c*)
12.4	\L'*Nc*'	Vertical line drawing function (optionally with *c*)
8	\n*z*,\n(*zz*	Interpolate number register *z* or *zz*
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*
-	**c.**	Input line-number in current input file; same as .c.
3	**%**	Current page number.
11.2	**ct**	Character type (set by *width* function).
7.4	**dl**	Width (maximum) of last completed diversion.
7.4	**dn**	Height (vertical size) of last completed diversion.
-	**dw**	Current day of the week (1-7).
-	**dy**	Current day of the month (1-31).
11.3	**hp**	Current horizontal place on *input* line.
15	**ln**	Output line number.
-	**mo**	Current month (1-12).
4.1	**nl**	Vertical position of last printed text base-line.
11.2	**sb**	Depth of string below base line (generated by *width* function).
11.2	**st**	Height of string above base line (generated by *width* function).
-	**yr**	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	**.$**	Number of arguments available at the current macro level.
-	**.A**	Set to 1 in *troff*, if **-a** option used; always 1 in *nroff*.
11.1	**.H**	Available horizontal resolution in basic units.
-	**.L**	Current line-spacing parameter (**ls**).
-	**.P**	1 if current page is printed, otherwise zero.
-	**.T**	Set to 1 in *nroff*, if **-T** option used; always 0 in *troff*.
11.1	**.V**	Available vertical resolution in basic units.
5.2	**.a**	Post-line extra line-space most recently utilized using **\x'*N*'**.
-	**.c**	Number of *lines* read from current input file.
7.4	**.d**	Current vertical place in current diversion; equal to **nl**, if no diversion.
2.2	**.f**	Current font as physical quadrant (1-4).
4	**.h**	Text base-line high-water mark on current page or diversion.
6	**.i**	Current indent.
-	**.j**	Current adjustment mode and type.
-	**.k**	Horizontal text portion size of current 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.

Table of Contents

List of Tables

Chapter 4

Formatting Tables with tbl

This chapter[1] provides instructions for preparing *tbl* input to format tables and for running the *tbl* preprocessor on a file. It also supplies numerous examples after which to pattern your own tables. The description of instructions is precise but technical, and the newcomer may prefer to glance over the examples first, as they show some common table arrangements.

Tbl turns a simple description of a table into a *troff* or *nroff* program that prints the table. From now on, unless noted specifically, we'll refer to both *troff* and *nroff* as *troff* since *tbl* treats them the same. *Tbl* makes phototypesetting tabular material relatively simple compared to normal typesetting methods. You may use *tbl* with the equation formatting program *eqn* or various layout macro packages, as *tbl* does not duplicate their functions.

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

The input to *tbl* is text for a document, with the text 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 formatting macros, can use these lines to delimit and place tables as necessary. In particular, any arguments on the '.TS' or '.TE' lines are copied but otherwise ignored, and may be used by document layout macro commands.

[1] The material in this chapter is derived from *Tbl — A Program to Format Tables*, M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey.

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, indicated by *format*, followed by the *data* to be entered in the table. You may precede the formatting information, which describes the individual columns and rows of the table, by *options* that affect the entire table.

4.1. Running tbl

You can run *tbl* on a simple table by piping the *tbl* output to *troff* (or your installation's equivalent for the phototypesetter) with the command:

```
logo% tbl file | troff -options
```

where *file* is the name of the file you want to format. For more complicated use, where there are several input files, and they contain equations and *-ms* macro package requests as well as tables, the normal command is:

```
logo% tbl file1 file2 . . . | eqn | troff -ms
```

You can, of course, use the usual options 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.

If you are running *tbl* on a line printer that does not filter reverse paper motions, use the *col* processor to filter the multicolumn output.

If you are using an IBM 1403 line printer 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*, *-mm*, and 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, *tbl* accepts them as well.

Caveats: Note that when you use *eqn* and *tbl* together on the same file, put *tbl* 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*, you must put *tbl* first or the output will be scrambled. 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. To avoid this, give the *delim (xx)* table option; this prevents splitting 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. Avoid using *troff* number registers used by *tbl* 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 that you may not use them 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. Use of this macro in the page footer boxes a multi-page table. In particular, you can use the *–ms* macros 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 there aren't any. 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. For example:

```
.TS H
center box tab (/);
c s
l l .
Employees
_
Name/Phone
_
.TH
Jonathan Doe/123-4567
< etc. >
.TE
```

Note that this is *not* a feature of *tbl*, but of the *–ms* layout macros.

4.2. Input Commands

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The sections that follow explain how to enter the various parts

of the table.

4.2.1. Options

There may be a single line of options affecting the whole table. If present, this line must follow the '.TS' line immediately, 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-adjusted)
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 (such as from **box**) in *n* point type
delim(*xy*)	recognize *x* and *y* as the *eqn* delimiters

A standard option line is:

```
center box tab (/) ;
```

which centers the table on the page, draws a box around it, and uses the slash '/' character as the column separator.

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, so if there are spacing commands embedded in the input, these requests may be inaccurate. Use normal *troff* procedures, such as keep-release macros, in this case. If you must have a multi-page boxed table, use macros designed for the purpose, as explained below under *Running 'tbl'*.

4.2.2. Format

The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table, except that the last line corresponds to all following lines up to the next '.T&', if present as shown below. 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, tabs, or a visible character such as a slash '/'. Each key-letter is one of the following:

L or **l**	indicates a left-adjusted column entry.
R or **r**	indicates a right-adjusted column entry.
C or **c**	indicates a centered column entry.
N or **n**	indicates a numerical column entry, to line up the units digits of numerical entries.
A or **a**	indicates an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see the 'Some London Transport Statistics' example).
S or **s**	indicates a spanned heading; that is, it indicates that the entry from the previous column continues across this column; not allowed for the first column.

^ indicates a vertically spanned heading; that is, it indicates that the entry from the previous row continues down through this row; not allowed for the first row of the table.

When you specify numerical alignment, *tbl* requires a location for the decimal point. 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, you may use the special non-printing character string '\&' 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 (we use **L** here 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 way **a** type data are formatted, 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.

Note: Do not use the **n** and **a** items in the same column.

For readability, separate the key-letters describing each column with spaces. Indicate the end of the format section 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 is:

```
.TS
c s s
l n n .
text
.TE
```

which specifies a table of three columns. The first line of the table contains a centered heading that spans 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 is:

	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

Additional features of the key-letter system follow:

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. You

can also type this in the data portion. 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 displayed.

Vertical lines
— A vertical bar may be placed between column key-letters. This draws 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').[2] If the 'expand' option is used, 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, that is the 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 begins 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, which is 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 no effect unless the corresponding table entry is a text block (see *Text Blocks* 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 considered to be that wide. If the largest element in the column is wider than the specified value, its

[2] More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

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 is used, the default is ens. If the width specification is a unitless integer, you may omit the parentheses. 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. In this way, you can format 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, separate successive line formats on the same line by commas. The format for the sample table above can be 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.

4.2.3. Data

Type the data for the table after the format line. Normally, each table line is typed as one line of data. Break very long input lines by typing a backslash ' \' as a continuation marker at the end of the run-on line. That line is combined with the following line upon formatting and the ' \' vanishes. The data for different columns, that is, the table entries, are separated by tabs, or by whatever character has been specified in the option *tab* option. We recommend using a visible character such as the slash character '/'. 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, you can produce space within a table by '.sp' commands in the data.

Full width horizontal lines
— An input *line* containing only the character '_' (underscore) or '=' (equal sign) represents 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 '=' represents 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 '_' represents 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}'. To enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs, use:

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

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 'New York Area Rocks' example for an illustration of included text blocks in a table. If you use more than twenty or thirty text blocks in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many text block diversions.'

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.

Note:
Although you can put any number of lines in a table, only the first 200 lines are used in calculating the widths of the various columns. Arrange a multi-page table 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, use requests such as '.ps' with care to avoid confusing the width calculations.

4.2.4. Changing the Format

If you must change the format of a table after many similar lines, as with sub-headings or summarizations, use the '.T&' (table continue) command 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 'Composition of Foods' and 'Some London Transport Statistics' examples. Using this procedure, each table line can be close to its corresponding format line.

Note: 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.

4.3. Examples

Here are some examples illustrating features of *tbl*. Glance through them to find one that you can adapt to your needs.

Although you can use a tab to separate columns of data, a visible character is easier to read. The standard column separator here is the slash '/'. If a slash is part of the data, we indicate a different separator, as in the first example.

Input:

```
.TS
 tab (%) 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
tab (/) 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
tab (/) 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
tab (/) ;
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
tab (/) 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	Months
june	july	
august	september	
october	november	december

Input:

```
.TS
 tab (/) 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
 tab (/) 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
tab (/) 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	$\sin(z)=\frac{1}{2i}(e^{iz}-e^{-iz})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^\pi \cos(z\sin\theta)\,d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty} k^{-s}$ (Re $s>1$)

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 & Leading for 10-Pt. 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 & Leading for 10-Pt. 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
tab (/) ;
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./8,553
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
.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.	5,553
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

Input:

```
.ps 8
.vs 10p
.TS
tab (/) 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. Examine the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
center tab (/) ;
c s s s
c s s s
c c c c
n n n n.
LYKE WAKE WALK
Successful Crossings 1959–1966
Year/First Crossings/Repeats/Total
1959/89/23/112
1960/222/33/255
1961/650/150/800
1962/1100/267/1367
1963/1054/409/1463
1964/1413/592/2005
1965/2042/771/2813
1966/2537/723/3260
.TE
```

Output:

LYKE WAKE WALK
Successful Crossings 1959–1966

Year	First Crossings	Repeats	Total
1959	89	23	112
1960	222	33	255
1961	650	150	800
1962	1100	267	1367
1963	1054	409	1463
1964	1413	592	2005
1965	2042	771	2813
1966	2537	723	3260

Input:

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

_
Name/Description/Practical Information

_
T{
American Museum of Natural History
T}/T{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors.  There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}/Hours/10-5, ex. Sun 11-5, Wed. to 9
\^/\^/Location/T{
Central Park West & 79th St.
T}
\^/\^/Admission/Donation: $1.00 asked
\^/\^/Subway/AA to 81st St.
\^/\^/Telephone/212-873-4225

_
Bronx Zoo/T{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}/Hours/T{
10-4:30 winter, to 5:00 summer
T}
\^/\^/Location/T{
185th St. & Southern Blvd, the Bronx.
T}
\^/\^/Admission/$1.00, but Tu,We,Th free
\^/\^/Subway/2, 5 to East Tremont Ave.
\^/\^/Telephone/212-933-1759

_
Brooklyn Museum/T{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}/Hours/Wed-Sat, 10-5, Sun 12-5
\^/\^/Location/T{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^/\^/Admission/Free
\^/\^/Subway/2,3 to Eastern Parkway.
\^/\^/Telephone/212-638-5000

_
T{
New-York Historical Society
T}/T{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}/Hours/T{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^/\^/Location/T{
Central Park West & 77th St.
T}
\^/\^/Admission/Free
\^/\^/Subway/AA to 81st St.
\^/\^/Telephone/212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: $1.00 asked AA to 81st St. 212-873-4225
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. $1.00, but Tu,We,Th free 2, 5 to East Tremont Ave. 212-933-1759
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

4.4. Tbl Commands

Table 4-1: *tbl* Command Characters and Words

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

Table of Contents

List of Tables

Chapter 5

Typesetting Mathematics with eqn

This chapter[1] explains how to use the *eqn* preprocessor for printing mathematics on a phototypesetter and provides numerous examples after which to model equations in your documents.

You describe mathematical expressions in an English-like language that the *eqn* program translates into *troff* commands for final *troff* formatting. In other words, *eqn* sets the mathematics while *troff* does the body of the text. *Eqn* provides accurate and relatively easy mathematical phototypesetting, which is not easy to accomplish with normal typesetting machines. Because the mathematical expressions are imbedded in the running text of a manuscript, the entire document is produced in one process. For example, you can set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

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

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

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

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 these symbols, and they will look good.

Eqn also produces mathematics with *nroff*. The input is identical, but you have to use the programs *neqn* instead of *eqn* and *troff*. Of course, some things won't look as good because your workstation or terminal does not provide the variety of characters, sizes and fonts that a phototypesetter does, but the output is usually adequate for proofreading.

5.1. Displaying Equations — '.EQ' and '.EN'

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

[1] The material in this chapter is derived from *A System for Typesetting Mathematics*, B.W. Kernighan, L. L. Cherry and *Typesetting Mathematics — User's Guide*, B.W. Kernighan, L.L. Cherry, Bell Laboratories, Murray Hill, New Jersey.

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

your output will look like:

$$x=y+z$$

Eqn copies '.EQ' and '.EN' through untouched. This means that you have to take care of things like centering, numbering, and so on yourself. The common way is to use the *troff* and *nroff* macro package package '–ms', which provides macros for centering, indenting, left-justifying and making numbered 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'.

You can also supplement *eqn* with *troff* commands as desired; for example, you can produce a centered display with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

which produces

$$x_i=y_i \cdots$$

You can call out any of these 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 you can enter in-line expressions like π_i^2 without '.EQ ' and '.EN'. This is described in *Shorthand for In-line Equations*.

5.2. Running eqn and neqn

To print a document that contains mathematics on the phototypesetter, use:

```
logo% eqn files | troff –options
logo%
```

where *troff* (or your installation's equivalent) sends the output to your phototypesetter. If you use the *–ms* macro package for example, type:

```
logo% eqn files | troff –ms
logo%
```

To display equations on the standard output, your workstation screen, use *nroff* as follows:

```
logo% neqn files | nroff –options
```

The language for equations recognized by *neqn* is identical to that of *eqn*, although of course the output is more restricted. You can use the online rendition of the mathematical formulae for proofing, but the output does not accurately represent the symbols and fonts. You can of course pipe the output through *more* for easier viewing:

logo% **neqn** *files* | **nroff** *-options* |**more**

or redirect it to a file:

logo% **neqn** *files* | **nroff** *-options* >*newfile*

To use a GSI or DASI terminal as the output device, type:

logo% **neqn** *files* | **nroff** **-T***x*

where is the terminal type you are using, such as *300* or *300S*. To send *neqn* output to the printer, type:

logo% **neqn** *file* | **nroff** *-options* | **lpr** **-P***printer*

You can use *eqn* and *neqn* with the *tbl* program for setting tables that contain mathematics. Use *tbl* before *eqn* or *neqn*, like this:

```
logo% tbl files | eqn | troff -options
logo%
```

or

logo% **tbl** *files* | **neqn** | **nroff** *-options*

5.3. Putting Spaces in the Input Text

Eqn throws away spaces and newlines within an expression and leaves normal text alone. Thus between '.EQ' and '.EN ',

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

and

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

and

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

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.

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. To do this, surround a special word by ordinary spaces (or tabs or newlines), as shown in the previous section.

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

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

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:

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

You can also use braces '{ }' and double quotes ' "..." ' to separate special words; these characters which have special meanings are described later.

Remembering that a blank is a delimiter can be a problem. For instance, a common mistake is typing:

```
.EQ
f(x sub i)
.EN
```

which produces

$$f(x_{i)}$$

instead of

$$f(x_i)$$

Eqn cannot tell that the right parenthesis is not part of the subscript. Type instead:

```
.EQ
f(x sub i )
.EN
```

5.4. Producing Spaces in the Output Text

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

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

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. Use tabs to position pieces of an expression, but you must use *troff* commands to set the tab stops.

5.5. Symbols, Special Names, and Greek Letters

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

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

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 *Precedences and Keywords*. You can also use special characters available in *troff* for anything *eqn* doesn't know about.

5.6. Subscripts and Superscripts — 'sub' and 'sup'

To obtain subscripts and superscripts, use the words *sub* and *sup*.

```
.EQ
x sup 2 + y sub k
.EN
```

gives

$$x^2+y_k$$

Eqn takes care of all the size changes and vertical motions needed to make the output look right. You must surround the words *sub* and *sup* by spaces; *x sub2* gives you *xsub*2 instead of x_2. As another example, consider:

```
.EQ
x sup 2 + y sup 2 = z sup 2
.EN
```

which produces:

$$x^2+y^2=z^2$$

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

```
.EQ
y = (x sup 2)+ 1
.EN
```

which causes

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

instead of the intended

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

which is produced by:

```
.EQ
y = (x sup 2 )+ 1
.EN
```

Subscripted subscripts and superscripted superscripts also work:

```
.EQ
x sub i sub 1
.EN
```

is

$$x_{i_1}$$

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

```
.EQ
x sub i sup 2
.EN
```

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 $x^y{}_z$.

5.7. Grouping Equation Parts — '{' and '}'

Normally, the end of a subscript or superscript is marked simply by a blank, tab, tilde, and so on. If the subscript or superscript is something that has to be typed with blanks in it, use the braces '{' and '}' to mark the beginning and end of the subscript or superscript:

```
.EQ
e sup {i omega t}
.EN
```

is

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

You can *always* use braces to force *eqn* to treat something as a unit, or just to make your intent perfectly clear. Thus:

```
.EQ
x sub {i sub 1} sup 2
.EN
```

is

$$x_{i_1}^2$$

with braces, but

```
.EQ
x sub i sub 1 sup 2
.EN
```

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces if necessary:

```
.EQ
e sup {i pi sup {rho + 1}}
.EN
```

is

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

The general rule is that anywhere you could use some single entry like x, you can use an arbitrarily complicated entry if you enclose it in braces. *Eqn* looks 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 causes *eqn* to complain bitterly.

Occasionally you have to print braces. To do this, enclose them in double quotes, like ' "{" '. Quoting is discussed in more detail in *Quoted Text*.

5.8. Fractions — 'over'

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

```
.EQ
a+ b over 2c =1
.EN
```

gives

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

The line is made the right length and positioned automatically.

```
.EQ
a+ b over c+ d+ e = 1
.EN
```

produces

$$\frac{a+b}{c+d+e}=1$$

Use braces to clarify what goes over what:

```
.EQ
{alpha + beta} over {sin (x)}
.EN
```

is

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

When there is both an *over* and a *sup* in the same expression, *eqn* does the *sup* before the *over*, so

```
.EQ
-b sup 2 over pi
.EN
```

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 *Precedences and Keywords*. When in doubt, however, use *braces* to make clear what goes with what.

5.9. Square Roots — 'sqrt'

To draw a square root, use *sqrt:*

```
.EQ
sqrt a+ b
.EN
```

produces

$$\sqrt{a+b}$$

and

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

is

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

Note: Square roots of tall quantities look sloppy because a root-sign big enough to cover the quantity is too dark and heavy:

```
.EQ
sqrt {a sup 2 over b sub 2}
.EN
```

is

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

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

$$(a^2/b_2)^{\frac{1}{2}}$$

which is

```
.EQ
(a sup 2 /b sub 2 ) sup {1 over 2}
.EN
```

5.10. Summation, Integral, and Other Large Operators

To produce summations, integrals, and similar constructions, use:

```
.EQ
sum from i=0 to {i= inf} x sub i
.EN
```

which produces

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

Notice that you use braces to indicate where the upper part $i=\infty$ begins and ends. No braces are necessary for the lower part $i=0$, because it does not contain any 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:

```
.EQ
int  prod  union  inter
.EN
```

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:

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

is

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

5.11. Size and Font Changes

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

```
.EQ
bold x y
.EN
```

is

$$\mathbf{x}y$$

and

```
.EQ
size 14 bold x == y +
  size 14 {alpha + beta}
.EN
```

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

```
.EQ
size 12 { ... }
.EN
```

Legal sizes which may follow *size* are the same as those allowed in *troff*: 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 is easier because 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 throughout a document: you can change the global font and size as often as needed. For example, in a footnote[2] you will typically want the size of equations to match the size of the footnote

[2] Like this one, in which we have a few random expressions like x_i and x^2. The sizes for these were set by the command *gsize~ -2*.

text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

5.12. 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	$\overline{x}$
x under	$\underline{x}$

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

```
.EQ
x dot under + x hat + y tilde
+ X hat + Y dotdot = z + Z bar
.EN
```

produces

$$\underline{\dot{x}} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \overline{z + Z}$$

5.13. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments that you normally set. This provides a way to do your own spacing and adjusting if needed:

```
.EQ
italic "sin(x)" + sin (x)
.EN
```

is

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

You also use quotes to get braces and other *eqn* keywords printed:

```
.EQ
"{ size alpha }"
.EN
```

is

$$\{\ size\ alpha\ \}$$

and

```
.EQ
roman "{ size alpha }"
.EN
```

is

$$\text{roman } \{ \text{ size alpha } \}$$

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

```
.EQ
"" sup 2 roman He
.EN
```

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.

5.14. Lining Up Equations — 'mark' and 'lineup'

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. To do this, use the 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 out of the scope of this chapter, when you use *eqn* and '*-ms*', use either '.EQ I' or '.EQ L', as *mark* and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead;

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

isn't going to work, because there isn't room for the $x+ y$ part after the *mark* has processed the z.

5.15. Big Brackets

To get big brackets '[ ]', braces '{ }', parentheses '( )', and bars '| |' around things, use the *left* and *right* commands:

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

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 they are not likely to look very good. One exception is the *floor* and *ceiling* characters:

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

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.

5.16. Piles — 'pile'

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

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

will make

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

The elements of the pile are centered one above another at the right height for most purposes. There can be as many elements as you want. The keyword *above* is used to separate the pieces; put braces 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. For example:

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

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.

5.17. Matrices — 'matrix'

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

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

you have to type

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

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***. Otherwise, results are unpredictable.

5.18. Shorthand for In-line Equations — 'delim'

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 you need variable names like x to be in italics. Although you can do this 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 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 $sum from i=1 to n x sub i$ does not interfere with the lines surrounding it.

The printed result looks like: Let α_i be the primary variable, and let β be zero. Then we can show that x_1 is ≥ 0.

To turn off the delimiters, use:

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

Note: Don't use braces, tildes, circumflexes, or double quotes as delimiters; chaos will result.

5.19. Definitions — 'define'

Eqn provides a string-naming 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

```
.EQ
x sub i sub 1 + y sub i sub 1
.EN
```

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

```
.EQ
define xy 'x sub i sub 1 + y sub i sub 1'
.EN
```

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 sure 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

```
.EQ
define X ' roman X '
.EN
```

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

```
.EQ
define X ' roman "X" '
.EN
```

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

You can redefine *eqn* keywords. You can make '/' mean *over* by saying

```
.EQ
define / ' over '
.EN
```

or redefine *over* as '/' with

```
.EQ
define over ' / '
.EN
```

If you need things to print on a workstation or terminal as well as on the phototypesetter, it is sometimes worth defining a symbol differently in *neqn* and *eqn*. To do this, use *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*.

5.20. Tuning the Spacing

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. You can get small extra horizontal spaces with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. The *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 anything if it is enclosed in braces.

5.21. Troubleshooting

If you make a mistake in an equation, like leaving out a brace, having one too many, or having a *sup* with nothing before it, *eqn* tells 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 nonexistent file.

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

```
logo% eqn files >/dev/null
```

to throw away the output but display the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. You may also occasionally forget one half of a pair of macros or have an unbalanced font change. These can cause problems, but you can check for balanced pairs of delimiters and macros with *checkeq* and *checknr*. For instance, to run *checkeq* on this chapter called *eqn.ug* to check for unbalanced pairs of '.EQ' and '.EN', type:

```
logo% checkeq eqn.ug
eqn.ug:
   New delims , line 2
    in EQ, line 2
   Spurious EN, line 46
   Delim off, line 1254
   New delims , line 1278
   New delims , line 1635
    in EQ, line 1635
   New delims ##, line 1991
   Delim off, line 1999
logo%
```

We left out the '.EQ' before the '.EN' on line 46 to show you some sample output. This also

reports on the delimiters. You can also use *checknr* with specific options to check specifically for a particular macro pair. For example, to run *checknr* to check that there is an '.EQ' for every '.EN', type:

```
logo% checknr -s -f -a.EQ.EN eqn.ug
46: Unmatched .EN
logo%
```

Specify the macro pair you want to check for with the **-a** option and the six characters in the pair. The **-s** option ignores size changes and the *-f* option ignores font changes. See the user's manual on *checknr* for more details.

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, that is, offset from the body of the text with '.EQ' and '.EN', 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.

5.22. Precedences and Keywords

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
```

The operations that group to the left are:

```
over sqrt left right
```

All others group to the right. For example, in the expression

```
.EQ
a sup 2 over b
.EN
```

sup is defined to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$. Naturally, you can always force a particular parsing by placing braces around expressions.

Digits, parentheses, brackets, punctuation marks, and the following 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
```

The following character sequences are recognized and translated as shown.

Table 5-1: Character Sequence Transalation

You Type	Translation
>=	≥
<=	≤
==	≡
!=	≠
+-	±
->	→
<-	←
<<	≪
>>	≫
inf	∞
partial	∂
prime	′
approx	≈
nothing	
cdot	·
times	×
del	∇
grad	∇
...	. . .
,....,	, . . . ,
sum	∑
int	∫
prod	∏
union	∪
inter	∩

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

Table 5-2: Greek Letters

You Type	Translation	You Type	Translation
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		

The *eqn* keywords, except for characters with names, follow.

Table 5-3: *eqn* Keywords

above	lpile
back	mark
bar	matrix
bold	ndefine
ccol	over
col	pile
cpile	rcol
define	right
delim	roman
dot	rpile
dotdot	size
down	sqrt
dyad	sub
fat	sup
font	tdefine
from	tilde
fwd	to
gfont	under
gsize	up
hat	vec
italic	~ ^ ,
lcol	{ }
left	" ... "
lineup	

5.23. Several Examples

Here is the complete source for several examples and for the three display equations in the introduction to this chapter.

Squareroot

Input:

```
.EQ
x = {-b + - sqrt{b sup 2-4ac}} over 2a
.EN
```

Output:

$$x = \frac{-b + -\sqrt{b^{2-4ac}}}{2a}$$

Summation, Integral, and Other Large Operators

Input:

```
.EQ
lim from {x -> pi /2} ( tan~x) = inf
.EN
```

Output:

$$\lim_{x \to \pi/2} (\tan\ x) = \infty$$

Input:

```
.EQ
sum from i=0 to infinity x sub i = pi over 2
.EN
```

Output:

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

Input:

```
.EQ
lim from {x-> pi /2} ( tan~x) sup{sin~2x}~=~1
.EN
```

Output:

$$\lim_{x \to \pi/2} (\tan\ x)^{\sin 2x} = 1$$

Input:

```
.EQ
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 )
}
.EN
```

Output:

$$\int \frac{dx}{ae^{mx}-be^{-mx}} = \begin{cases} \frac{1}{2m\sqrt{ab}} \log \frac{\sqrt{a}e^{mx}-\sqrt{b}}{\sqrt{a}e^{mx}+\sqrt{b}} \\ \frac{1}{m\sqrt{ab}} \tanh^{-1}(\frac{\sqrt{a}}{\sqrt{b}}e^{mx}) \\ \frac{-1}{m\sqrt{ab}} \coth sup-1 \frac{(sa}{\sqrt{b}}e^{mx}) \end{cases}$$

Quoted Text

Input:

```
.EQ
lim~ roman "sup" ~x sub n = 0
.EN
```

Ouput:

$$\lim \sup x_n = 0$$

Big Brackets

Input:

```
.EQ
left [ x+ y over 2a right ]~=~1
.EN
```

Output:

$$\left[\frac{x+y}{2a}\right] = 1$$

Fractions

Input:

```
.EQ
a sub 0 + b sub 1 over
 {a sub 1 + b sub 2 over
  {a sub 2 + b sub 3 over
   {a sub 3 + ...}}}
.EN
```

Output:

$$a_0+\cfrac{b_1}{a_1+\cfrac{b_2}{a_2+\cfrac{b_3}{a_3+\cdots}}}$$

Input:

```
.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
```

Output:

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

Input:

```
.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
```

Output:

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

Input:

```
.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
```

Output:

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

Shorthand for In-line Equations

Input:

```
.EQ
delim ##
.EN
```

Let #x sub i#, #y# and #alpha# be positive

Output:

Let x_i, y and α be positive

Table of Contents

Chapter 6

Making Bibliographic References with refer

This chapter[1] describes *refer*, the preprocessor for *nroff* and *troff* that finds and formats bibliographic references. There is a user information for *refer* in *Selecting and Formatting References for 'troff'*. *Reference Files* details reference files for adding references to data bases or writing new *troff* macros to use with *refer*. The options to make *refer* collect identical citations, or otherwise relocate and adjust references, are described in *Collecting References and other 'refer' Options*.

This chapter is of interest to those who are interested in facilities for searching large but relatively unchanging text files on the Sun system, and those who are interested in handling bibliographic citations with *troff*. The Sun system has many utilities, such as, *grep, awk, lex, egrep,* and *fgrep,* that search through files of text, but most of them are based on a linear scan through the entire file. *Refer* uses inverted indexes so you can use it on much larger data bases. See D. Knuth, *The Art of Computer Programming: Vol. 3, Sorting and Searching,* Addison-Wesley, Reading, Mass., (1977). See section 6.5. for more information.

Refer draws from a list of 4300 references that is maintained on line and contains primarily papers written and cited by local authors. Whenever one of these references is required in a paper, a few words from the title or author list will retrieve it, and you need not bother to re-enter the exact citation. Alternatively, authors can use their own lists of papers.

There are also auxiliary programs to update reference lists. The programs permit a large amount of individual control over the content of publication lists, but retain the usefulness of the files to other users. *refer* uses standard input by default as its input file, and copies it to standard output. You can also update the reference lists.

Macro packages print the finished reference text, flag the point of reference. References are noted by numbers by default. The way the system works is to make keyword indexes for volumes of material too large for linear searching. You can search quickly for combinations of single words. The programs for general searching are divided into two phases. The first makes an index from the original data, and the second searches the index and retrieves items. Both of these phases are further divided into two parts to separate the data-dependent and algorithm-dependent code. The *refer* preprocessor formats references, and the *lookall* command searches through all text files on the system.

6.1. Indexing and Searching

The indexing and searching process is divided into two phases, each made of two parts. These are:

---

[1] The material in this chapter is derived from *Updating Publication Lists*, M.E. Lesk and *Some Applications of Inverted Indexes on the UNIX System*, M.E. Lesk, Bell Laboratories, Murray Hill, New Jersey. It is in the process of being rewritten for the Sun system and may not yet accurately describe the program.

1. Construct the index.
   - Find keys -- turn the input files into a sequence of tags and keys, where each tag identifies a distinct item in the input and the keys for each such item are the strings under which it is to be indexed.
   - Hash and sort -- prepare a set of inverted indexes from which, given a set of keys, the appropriate item tags can be found quickly.
2. Retrieve an item in response to a query.
   - Search -- Given some keys, look through the files prepared by the hashing and sorting facility and derive the appropriate tags.
   - Deliver -- Given the tags, find the original items. This completes the searching process.

The first phase, making the index, is presumably done relatively infrequently. It should, of course, be done whenever the data being indexed change. In contrast, the second phase, retrieving items, is presumably done often, and must be rapid.

An effort is made to separate code which depends on the data being handled from code which depends on the searching procedure. The search algorithm is involved only in programs (hash and sort) and (search), while knowledge of the actual data files is needed only by programs (find keys) and (deliver). Thus it is easy to adapt to different data files or different search algorithms.

To start with, it is necessary to have some way of selecting or generating keys from input files. For dealing with files that are basically English, we have a key-making program which automatically selects words and passes them to the hashing and sorting program. The format used has one line for each input item, arranged as follows:

name:start,length (tab) key1 key2 key3 ...

where *name* is the file name, *start* is the starting byte number, and *length* is the number of bytes in the entry.

These lines are the only input used to make the index. The first field (the file name, byte position, and byte count) is the tag of the item and can be used to retrieve it quickly. Normally, an item is either a whole file or a section of a file delimited by blank lines. After the tab, the second field contains the keys. The keys, if selected by the automatic program, are any alphanumeric strings which are not among the 100 most frequent words in English and which are not entirely numeric (except for four-digit numbers beginning 19, which are accepted as dates). Keys are truncated to six characters and converted to lower case. Some selection is needed if the original items are very large. We normally just take the first *n* keys, with *n* less than 100 or so; this replaces any attempt at intelligent selection. One file in our system is a complete English dictionary; it would presumably be retrieved for all queries.

To generate an inverted index to the list of record tags and keys, the keys are hashed and sorted to produce an index. What is wanted, ideally, is a series of lists showing the tags associated with each key. To condense this, what is actually produced is a list showing the tags associated with each hash code, and thus with some set of keys. To speed up access and further save space, a set of three or possibly four files is produced. These files are:

File	Contents
entry	Pointers to posting file for each hash code
posting	Lists of tag pointers for each hash code
tag	Tags for each item
key	Keys for each item (optional)

The posting file comprises the real data: it contains a sequence of lists of items posted under each hash code. To speed up searching, the entry file is an array of pointers into the posting file, one per potential hash code. Furthermore, the items in the lists in the posting file are not referred to by their complete tag, but just by an address in the tag file, which gives the complete tags. The key file is optional and contains a copy of the keys used in the indexing.

The searching process starts with a query, containing several keys. The goal is to obtain all items which were indexed under these keys. The query keys are hashed, and the pointers in the entry file used to access the lists in the posting file. These lists are addresses in the tag file of documents posted under the hash codes derived from the query. The common items from all lists are determined; this must include the items indexed by every key, but may also contain some items which are false drops, since items referenced by the correct hash codes need not actually have contained the correct keys. Normally, if there are several keys in the query, there are not likely to be many false drops in the final combined list even though each hash code is somewhat ambiguous. The actual tags are then obtained from the tag file, and to guard against the possibility that an item has false-dropped on some hash code in the query, the original items are normally obtained from the delivery program (deliver) and the query keys checked against them by string comparison.

Usually, therefore, the check for bad drops is made against the original file. However, if the key derivation procedure is complex, it may be preferable to check against the keys fed to program (hash and sort). In this case the optional key file which contains the keys associated with each item is generated, and the item tag is supplemented by a string

```
;start,length
```

which indicates the starting byte number in the key file and the length of the string of keys for each item. This file is not usually necessary with the present key-selection program, since the keys always appear in the original document.

There is also an option (**-C***n*) for coordination level searching. This retrieves items which match all but *n* of the query keys. The items are retrieved in the order of the number of keys that they match. Of course, *n* must be less than the number of query keys (nothing is retrieved unless it matches at least one key).

The *lookall* program is useful when looking for a document which you believe is stored on-line, but do not know where. For example, many memos can be in the file system, but it is often difficult to guess where a particular memo might be (it might have several authors, each with many directories, and have been worked on by a secretary with yet more directories). Instructions for the use of the *lookall* command are given in the manual section.

The only indexes maintained routinely are those of publication lists and all English files. To make other indexes, the programs for making keys, sorting them, searching the indexes, and delivering answers must be used. Since they are usually invoked as parts of higher-level commands, they are not in the default command directory, but are available to any user in the directory */usr/lib/refer*. Three programs are of interest: *mkey*, which isolates keys from input

files; *inv*, which makes an index from a set of keys; and *hunt*, which searches the index and delivers the items. Note that the two parts of the retrieval phase are combined into one program, to avoid the excessive system work and delay which would result from running these as separate processes.

These three commands have a large number of options to adapt to different kinds of input. If you not interested in the detailed description that now follows, skip to *Selecting and Formatting References for Troff*, which describes the *refer* program, the packaged-up version of these tools specifically oriented towards formatting references.

6.1.1. Make Keys — 'mkey'

The program *mkey* is the key-making program corresponding to the 'find keys' step in phase 1. Normally, it reads its input from the file names given as arguments, and if there are no arguments it reads from the standard input. It assumes that blank lines in the input delimit separate items, for each of which a different line of keys should be generated. The lines of keys are written on the standard output. Keys are any alphanumeric string in the input not among the most frequent words in English and not entirely numeric (except that all-numeric strings are acceptable if they are between 1900 and 1999). In the output, keys are translated to lower case, and truncated to six characters in length; any associated punctuation is removed. *Mkey* recognizes the following options:

mkey Options

Option	Meaning
-c *name*	Name of file of common words; default is */usr/lib/eign*.
-f *name*	Read a list of files from *name* and take each as an input argument.
-i *chars*	Ignore all lines which begin with '%' followed by any character in *chars*.
-k*n*	Use at most *n* keys per input item.
-l*n*	Ignore items shorter than *n* letters long.
-n*m*	Ignore as a key any word in the first *m* words of the list of common English words. The default is 100.
-s	Remove the labels *(file:start,length)* from the output; just give the keys. Used when searching rather than indexing.
-w	Each whole file is a separate item; blank lines in files are irrelevant.

The normal arguments for indexing references are the defaults, which are **-c /usr/lib/eign**, **-n100**, and **-l3**. For searching, the **-s** option is also needed. When the big *lookall* index of all English files is run, the options are **-w**, **-k50**, and **-f** *(filelist)*. When running on textual input, the *mkey* program processes about 1000 English words per processor second. Unless the **-k** option is used (and the input files are long enough for it to take effect) the output of *mkey* is comparable in size to its input.

6.1.2. Hash and Invert — 'inv'

The *inv* program computes the hash codes and writes the inverted files. It reads the output of *mkey* and writes the set of files described earlier in this section. It expects one argument, which is used as the base name for the three (or four) files to be written. Assuming an argument of *Index* (the default) the entry file is named *Index.ia*, the posting file *Index.ib*, the tag file *Index.ic*, and the key file (if present) *Index.id*. The *inv* program recognizes the following options:

inv Options

Option	Meaning
-a	Append the new keys to a previous set of inverted files, making new files if there is no old set using the same base name.
-d	Write the optional key file. This is needed when you can not check for false drops by looking for the keys in the original inputs, i.e. when the key derivation procedure is complicated and the output keys are not words from the input files.
-h*n*	The hash table size is *n* (default 997); *n* should be prime. Making *n* bigger saves search time and spends disk space.
-i[u] *name*	Take input from file *name*, instead of the standard input; if **u** is present *name* is unlinked when the sort is started. Using this option permits the sort scratch space to overlap the disk space used for input keys.
-n	Make a completely new set of inverted files, ignoring previous files.
-p	Pipe into the sort program, rather than writing a temporary input file. This saves disk space and spends processor time.
-v	Verbose mode; print a summary of the number of keys which finished indexing.

About half the time used in *inv* is in the contained sort. Assuming the sort is roughly linear, however, a guess at the total timing for *inv* is 250 keys per second. The space used is usually of more importance: the entry file uses four bytes per possible hash (note the **-h** option), and the tag file around 15-20 bytes per item indexed. Roughly, the posting file contains one item for each key instance and one item for each possible hash code; the items are two bytes long if the tag file is less than 65336 bytes long, and the items are four bytes wide if the tag file is greater than 65536 bytes long. Note that to minimize storage, the hash tables should be over-full; for most of the files indexed in this way, there is no other real choice, since the *entry* file must fit in memory.

6.1.3. Searching and Retrieving — 'hunt'

The *hunt* program retrieves items from an index. It combines, as mentioned above, the two parts of phase (2): search and delivery. The reason why it is efficient to combine delivery and search is partly to avoid starting unnecessary processes, and partly because the delivery operation must be a part of the search operation in any case. Because of the hashing, the search part takes place in two stages: first items are retrieved which have the right hash codes associated with them, and then the actual items are inspected to determine false drops, i.e. to determine if anything with the right hash codes doesn't really have the right keys. Since the original item is retrieved to check on false drops, it is efficient to present it immediately, rather than only giving the tag as output and later retrieving the item again. If there were a separate key file, this argument would not apply, but separate key files are not common.

Input to *hunt* is taken from the standard input, one query per line. Each query should be in *mkey* **-s** output format; all lower case, no punctuation. The *hunt* program takes one argument

which specifies the base name of the index files to be searched. Only one set of index files can be searched at a time, although many text files may be indexed as a group, of course. If one of the text files has been changed since the index, that file is searched with *fgrep;* this may occasionally slow down the searching, and care should be taken to avoid having many out of date files. *Hunt* recognizes the following option arguments:

***hunt* Options**

Option	Meaning
-a	Give all output; ignore checking for false drops.
-C*n*	Coordination level *n;* retrieve items with not more than *n* terms of the input missing; default *C0*, implying that each search term must be in the output items.
-F[yn*d***]**	**-Fy** gives the text of all the items found; **-Fn** suppresses them. **-F***d where d* is an integer gives the text of the first *d* items. The default is **-Fy**.
-g	Do not use *fgrep* to search files changed since the index was made; print an error comment instead.
-i *string*	Take *string* as input, instead of reading the standard input.
-l *n*	The maximum length of internal lists of candidate items is *n;* default 1000.
-o *string*	Put text output (**-Fy**) in *string;* of use *only* when invoked from another program.
-p	Print hash code frequencies; mostly for use in optimizing hash table sizes.
-T[yn*d***]**	**-Ty** gives the tags of the items found; **-Tn** suppresses them. **-T***d where d* is an integer gives the first *d* tags. The default is **-Tn**.
-t *string*	Put tag output (**-Ty**) in *string;* of use *only* when invoked from another program.

If you underspecify a query (one search term), many potential items will be examined and discarded as false drops, wasting time. If you overspecify the query (a dozen search terms), many keys will be examined only to verify that the single item under consideration has that key posted. The optimal search is achieved when the query just specifies the answer; however, overspecification is quite cheap. In general, overspecification can be recommended; it protects you against additions to the data base which turn previously uniquely-answered queries into ambiguous queries.

6.2. Selecting and Formatting References for 'troff' — 'refer' and 'lookbib'

Refer, is a *troff* preprocessor like *eqn* (see *Typesetting Mathematics with 'eqn'*) for processing mathematical equations. *Refer* scans its input looking for items of the form

```
.[
imprecise citation
.]
```

where an *imprecise citation* is merely a string of words found in the relevant bibliographic citation. This is translated into a properly formatted reference. If the imprecise citation does not correctly identify a single paper (either selecting no papers or too many) a message is given. The data base of citations searched may be tailored to each system, and individual users may specify their own citation files. The default data base is accumulated from the publication lists of the members of an organization, plus about half a dozen personal bibliographies that were collected.

For example, you can specify a reference for the paper by D. Knuth above as

```
...
sorting and searching
.[
knuth addison 1977
.]
...
```

The above input text can be processed by *refer* as well as *tbl* and *troff* with:

```
logo% refer memofile | tbl | troff -ms
```

and the reference is automatically translated into a correct citation to the paper on computer programming.

To place a reference in a paper using *refer*, first use the *lookbib* command to check that the paper is in the data base and to find out what keys are necessary to retrieve it. Type *lookbib* and then type some potential queries until a suitable query is found. For example, if you had one started to find a paper on *eqn* by presenting the query

```
logo% lookbib
kernighan cherry
(EOT)
```

lookbib would find several items. The query given above is adequate. Overspecifying the query is of course harmless. There are supplemental common extra keywords in the data base, such as common abbreviations for journals or other sources, to aid in searching.

If the reference is in the data base, you can insert the query that retrieved it in the text, between .[and .] brackets. If it is not in the data base, you can type it into a private file of references, using the format discussed in the next section, and then the -p option used to search this private file. Such a command might read (if the private references are called *myfile*)

```
logo% refer -p myfile document | tbl | eqn | troff -ms . . .
```

where you can omit *tbl* and/or *eqn* if they are not needed. You can use the *-ms* macros (see *Formatting Documents with the '-ms' Macro Package)* or some other macro package, but a macro package is essential. *Refer* only generates the data for the references; some macro package does the formatting, and if none is supplied the references will not be printed.

By default, the references are numbered sequentially, and the *-ms* macros format references as footnotes at the bottom of the page. Other possibilities are discussed in *Collecting References and Other 'refer' Options.*

6.3. Reference Files

A *reference file* is a set of bibliographic references that you can use with *refer*. It can be indexed using the software described in *Indexing and Searching* for fast searching. What *refer* does is to read the input document stream, looking for imprecise citation references. It then searches through reference files to find the full citations, and inserts them into the document. The format of the full citation is arranged to make it convenient for a macro package, such as the *-ms* macros, to format the reference for printing. Since the format of the final reference is determined by the desired style of output, which is determined by the macros used, *refer* avoids forcing any kind of reference appearance. All it does is define a set of string registers which contain the basic information about the reference; and provide a macro call which is expanded by the

macro package to format the reference. It is the responsibility of the final macro package to see that the reference is actually printed; if no macros are used, and the output of *refer* fed untranslated to *troff*, nothing at all will be printed.

The strings that *refer* defines are taken directly from the files of references, which are in the following format. Separate the references by blank lines. Each reference is a sequence of lines beginning with %% and followed by a key-letter. The remainder of that line, and successive lines until the next line beginning with %%, contain the information specified by the key-letter. In general, *refer* does not interpret the information, but merely presents it to the macro package for final formatting. If you have a separate macro package, for example, you can add new key-letters or use the existing ones for other purposes without bothering *refer*.

The meaning of the key-letters given below, in particular, is that assigned by the *-ms* macros. Not all information, obviously, is used with each citation. For example, if a document is both an internal memorandum and a journal article, the macros ignore the memorandum version and cite only the journal article. Some kinds of information are not used at all in printing the reference; if you do not like finding references by specifying title or author keywords, and prefer to add specific keywords to the citation, a field is available which is searched but not printed (**K**).

The key letters that *refer* and *-ms* currently recognize with the kind of information implied, are:

Key Letters

Key	Information Specified	Key	Information Specified
A	Author's name	N	Issue number
B	Title of book containing item	O	Other information
C	City of publication	P	Page(s) of article
D	Date	R	Technical report reference
E	Editor of book containing item	T	Title
G	Government (NTIS) ordering number	V	Volume number
I	Issuer (publisher)		
J	Journal name		
K	Keys (for searching)	X	or
L	Label	Y	or
M	Memorandum label	Z	Information not used by *refer*

For example, a sample reference is:

```
%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%Z ctr127
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. ACM
%V 23
%N 1
%P 1-12
%M abcd-78
%D Jan. 1976
```

Order is irrelevant, except that authors are shown in the order given. The output of *refer* is a stream of string definitions, one for each of the fields of each reference, as shown below.

```
.]-
.ds [A authors' names ...
.ds [T title ...
.ds [J journal ...
...
.][ type-number
```

The special macro .]– precedes the string definitions and the special macro .][follows. These are changed from the input .[and .] so that running the same file through *refer* again is harmless. The macro package uses the .]– macro for initialization. Use the .][macro to print the reference; add an argument *type-number* to indicate the kind of reference, as follows:

Argument Type-numbers

Value	Kind of reference
1	Journal article
2	Book
3	Article within book
4	Technical report
5	Bell Labs technical memorandum
0	Other

The reference is flagged in the text with the sequence

```
\*([.number\*(.]
```

where *number* is the footnote number. The macro package uses the strings [. and .] to format the reference flag in the text. These strings can be replaced for a particular footnote, as described in *Collecting References and other 'refer' Options*. The footnote number (or other signal) is available to the reference macro .][as the string register [F.

In some cases you may wish to suspend the searching, and merely use the reference macro formatting; that is, if you don't want to provide a search key between .[and .] brackets, but merely the reference lines for the appropriate document. Alternatively, you may wish to add a few fields to those in the reference as in the standard file, or to override some fields. To alter or replace fields, or supply whole references, insert lines beginning with %; any such line is taken as direct input to the reference processor rather than keys to be searched. Thus

```
.[
key1 key2 key3 ...
%Q New format item
%R Override report name
.]
```

makes the indicated changes to the result of searching for the keys. Put all of the search keys before the first % line.

If no search keys are provided, an entire citation can be provided in-line in the text. For example, if the *eqn* paper citation were to be inserted in this way, rather than by searching for it in the data base, the input would read

```
...
preprocessor like
.I eqn.
.[
%A B. W. Kernighan
%A L. L. Cherry
%T A System for Typesetting Mathematics
%J Comm. ACM
%V 18
%N 3
%P 151-157
%D March 1975
.]
It scans its input looking for items
...
```

This would produce a citation of the same appearance as that resulting from the file search.

As shown, fields are normally turned into *troff* strings. Sometimes you would rather have them defined as macros, so that you can put other *troff* commands into the data. When this is necessary, simply double the control character **%** in the data. Thus *refer* processes the input:

```
.[
%V 23
%%M
Bell Laboratories,
Murray Hill, N.J. 07974
.]
```

into

```
.ds [V 23
.de [M
Bell Laboratories,
Murray Hill, N.J. 07974
..
```

The information after **%%M** is defined as a macro to be invoked by **.[M** while the information after **%V** is turned into a string to be invoked by ***([V**. At present *-ms* expects all information as strings.

6.4. Collecting References and other Refer Options

Normally, the combination of *refer* and *-ms* prints output as *troff* footnotes, which are consecutively numbered and placed at the bottom of the page. However, there are options to place the references at the end, to arrange references alphabetically by senior author, and to indicate references by strings in the text of the form [Name1975a] rather than by number. Whenever references are not placed at the bottom of a page identical references are coalesced.

For example, the **-e** option to *refer* specifies that references are to be collected; in this case they are produced whenever the sequence

```
.[
$LIST$
.]
```

is encountered. Thus, to place references at the end of a paper, you run *refer* with the **–e** option and place the above **$LIST$** commands after the last line of the text. *Refer* then moves all the references to that point. To aid in formatting the collected references, *refer* writes the references preceded by the line

```
.]<
```

and followed by the line

```
.]>
```

to invoke special macros before and after the references.

Another *refer* option is **–s** which specifies sorting of references. The default, of course, is to list references in the order presented. The **–s** option implies the **–e** option, and thus requires a

```
.[
$LIST$
.]
```

entry to call out the reference list. The **–s** option may be followed by a string of letters, numbers, and '+' signs indicating how the references are to be sorted. The sorting is done using the fields whose key-letters are in the string as sorting keys; the numbers indicate how many of the fields are to be considered, with '+' taken as a large number. Thus the default is **–sAD** meaning 'Sort on senior author, then date.' To sort on all authors and then title, specify **–sA+T**. And to sort on two authors and then the journal, write **–sA2J**.

Other options to *refer* change the signal or label inserted in the text for each reference. Normally these are just sequential numbers, and the macro package determines their exact placement (within brackets, as superscripts, etc.). The **–l** option replaces reference numbers by strings composed of the senior author's last name, the date, and a disambiguating letter. If a number follows the **l** as in **–l3** only that many letters of the last name are used in the label string. To abbreviate the date as well the form **-l*m*,*n*** shortens the last name to the first *m* letters and the date to the last *n* digits. For example, the option **–l3,2** can refer to the *eqn* paper (reference 3) by the signal *Ker75a*, since it is the first cited reference by Kernighan in 1975.

If you want to specify particular labels for a private bibliography, use the **–k** option. Specifying **–k*x*** uses the field *x* as a label. The default is **L**. If this field ends in –, that character is replaced by a sequence letter; otherwise the field is used exactly as given.

If you want to suppress the *refer*-produced signals, use the **–b** option, which entirely suppresses automatic text signals.

You can override the *-ms* treatment of the reference signal, which is normally to enclose the number in brackets in *nroff* and make it a superscript in *troff*. If the lines **.[** or **.]** contain anything following these characters, the remainders of these lines are used to surround the reference signal, instead of the default. Thus, for example, to say 'See reference (2).' and avoid 'See reference.[2]' the input can be

```
See reference
.[(
imprecise citation ...
.]).
```

Note that blanks are significant in this construction. If a permanent change is desired in the style of reference signals, however, it is probably easier to redefine the strings **[.** and **.]**, which are used to bracket each signal, than to change each citation.

Although normally *refer* limits itself to retrieving the data for the reference, and leaves to a macro package the job of arranging that data as required by the local format, there are two special options for rearrangements that macro packages cannot do. The **–c** option puts fields into all upper case (CAPS-SMALL CAPS in *troff* output). The key-letters indicating what information is to be translated to upper case follow the **c**, so that **–cAJ** means that authors' names and journals are to be in caps. The **–a** option writes the names of authors last name first, that is *A. D. Hall, Jr.* is written as *Hall, A. D. Jr*. The citation form of the *Journal of the ACM*, for example, would require both **–cA** and **–a** options. This produces authors' names in the style *KERNIGHAN, B. W. AND CHERRY, L. L.* for the previous example. The **–a** option may be followed by a number to indicate how many author names should be reversed; **–a1** (without any **–c** option) would produce *Kernighan, B. W. and L. L. Cherry*, for example.

Finally, there is also the previously-mentioned **–p** option to specify a private file of references to be searched before the public files. Note that *refer* does not insist on a previously made index for these files. If a file is named which contains reference data but is not indexed, it will be searched (more slowly) by *refer* using *fgrep*. In this way it is easy for you to keep small files of new references, which you can add later to the public data bases.

6.5. Updating Publication Lists

This section describes several commands to update the publication lists. The data base consisting of these lists is kept in a set of files in the directory */usr/dict/papers*. The reason for having special commands to update these files is that they are indexed, and the only reasonable way to find the items to be updated is to use the index. However, altering the files destroys the usefulness of the index, and makes further editing difficult. So the recommended procedure is to

1. Prepare additions, deletions, and changes in separate files.
2. Update the data base and reindex.

Whenever you make changes or whatever, it is necessary to run the 'add & index' step changes do not take effect. The next section shows the format of the files in the data base. After that, the procedures for preparing additions, preparing changes, preparing deletions, and updating the public data base are given.

6.5.1. Publication Format

The format of a data base entry is summarized here through a few examples. In each example, the output format for an item is shown first, and then the corresponding data base entry.

Journal article:

A. V. Aho, D. J. Hirschberg, and J. D. Ullman, "Bounds on the Complexity of the Maximal Common Subsequence Problem," *J. Assoc. Comp. Mach.*, vol. 23, no. 1, pp. 1-12 (Jan. 1976).

```
%T Bounds on the Complexity of the Maximal Common
Subsequence Problem
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. Assoc. Comp. Mach.
%V 23
%N 1
%P 1-12
%D Jan. 1976
%M TM 75-1271-7
```

Conference proceedings:

B. Prabhala and R. Sethi, "Efficient Computation of Expressions with Common Subexpressions," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 222-230, Tucson, Ariz. (January 1978).

```
%A B. Prabhala
%A R. Sethi
%T Efficient Computation of Expressions with
Common Subexpressions
%J Proc. 5th ACM Symp. on Principles
of Programming Languages
%C Tucson, Ariz.
%D January 1978
%P 222-230
```

Book:

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass. (1976).

```
%T Software Tools
%A B. W. Kernighan
%A P. J. Plauger
%I Addison-Wesley
%C Reading, Mass.
%D 1976
```

Article within book:

J. W. de Bakker, "Semantics of Programming Languages," pp. 173-227 in *Advances in Information Systems Science, Vol. 2*, ed. J. T. Tou, Plenum Press, New York, N. Y. (1969).

```
%A J. W. de Bakker
%T Semantics of programming languages
%E J. T. Tou
%B Advances in Information Systems Science, Vol. 2
%I Plenum Press
%C New York, N. Y.
%D 1969
%P 173-227
```

Technical Report:

F. E. Allen, "Bibliography on Program Optimization," Report RC-5767, IBM T. J. Watson Research Center, Yorktown Heights, N. Y. (1975).

```
%A F. E. Allen
%D 1975
%T Bibliography on Program Optimization
%R Report RC-5767
%I IBM T. J. Watson Research Center
%C Yorktown Heights, N. Y.
```

Technical Memorandum:

A. V. Aho, B. W. Kernighan and P. J. Weinberg, "AWK – Pattern Scanning and Processing Language", TM 77-1271-5, TM 77-1273-12, TM 77-3444-1 (1977).

```
%T AWK – Pattern Scanning and Processing Language
%A A. V. Aho
%A B. W. Kernighan
%A P. J. Weinberger
%M TM 77-1271-5, TM 77-1273-12, TM 77-3444-1
%D 1977
```

You can enter other forms of publication similarly. Note that conference proceedings are entered as if journals, with the conference name on a %J line. This is also sometimes appropriate for obscure publications such as series of lecture notes. When something is both a report and an article, or both a memorandum and an article, enter all necessary information for both; see the first article above, for example. Extra information (such as "In preparation" or "Japanese translation") should be placed on a line beginning %O . The most common use of %O lines now is for "Also in ..." to give an additional reference to a secondary appearance of the same paper.

Note that %B indicates the title of a book containing the article being entered; when an item is an entire book, enter the title with a %T as usual.

Normally, the order of items does not matter. The only exception is that if there are multiple authors (%A lines), the order of authors should be that on the paper. If a line is too long, you

can continue it on to the next line; any line not beginning with % or '.' (dot) is assumed to be a continuation of the previous line. Again, see the first article above for an example of a long title. Except for authors, do not repeat any items; if two %J lines are given, for example, the first is ignored. Separate multiple items on the same file by blank lines.

Note that in formatted printouts of the file, the exact appearance of the items is determined by a set of macros and the formatting programs. Do not try to adjust fonts, punctuation, or whatever by editing the data base. If someone has a real need for a differently-formatted output, generate a new set of macros to provide alternative appearances of the citations.

6.5.2. Updating and Re-indexing

This section describes the commands for manipulating and changing the data base. It explains the procedures for (a) finding references in the data base, (b) adding new references, (c) changing existing references, and (d) deleting references. Remember that you should prepare all changes, additions, and deletions in separate files and then run an 'update and reindex' step.

6.5.2.1. Checking What's There Now

Often you will want to know what is currently in the data base. There is a special command *lookbib* to look for things and print them out. It searches for articles based on words in the title, or the author's name, or the date. For example, you could find the first paper above with

```
logo% lookbib aho ullman maximal subsequence 1976
```

or

```
logo% lookbib aho ullman hirschberg
```

If you don't give enough words, several items will be found; if you spell some wrong, nothing will be found. There are around 4300 papers in the public file; you should always use this command to check when you are not sure whether a certain paper is there or not.

6.5.2.2. Adding New Papers

To add new papers, just type in, on one or more files, the citations for the new papers. Remember to check first if the papers are already in the data base. For example, if a paper has a previous memo version, this should be treated as a change to an existing entry, rather than a new entry. If several new papers are being typed on the same file, be sure that there is a blank line between each two papers.

6.5.2.3. Changing Items

To change an item, extract it onto a file with the command

```
logo% pub.chg key1 key2 key3 ...
```

where the items key1, key2, key3, etc. are a set of keys that will find the paper, as in the *lookbib* command. That is, if

```
logo% lookbib johnson yacc cstr
```

finds a item (in this case, Computing Science Technical Report No. 32, "YACC: Yet Another Compiler-Compiler," by S. C. Johnson), then

logo% **pub.chg johnson yacc cstr**

permits you to edit the item. The ***pub.chg*** command extracts the item onto a file named ***bibzzz*** where ***zzz*** is a 3-digit number, such as ***bib234***. The command displays the filename it has chosen. If the set of keys finds more than one paper (or no papers), an error message is displayed and no file is written. You must extract each reference to be changed with a separate ***pub.chg*** command, so each will be placed on a separate file. You should then edit the ***bibzzz*** file as desired to change the item, using an editor. Do not delete or change the first line of the file, however, which begins *%#* as it is a special code line to tell the update program which item is being altered. You may delete or change other lines, or add lines, as you wish. The changes are not actually made in the public data base until you run the update command ***pub.run*** (described below). Thus, if after extracting an item and modifying it, you decide that you'd rather leave things as they were, delete the ***bibzzz*** file, and your change request will disappear.

6.5.2.4. Deleting Entries

To delete an entry from the data base, type:

logo% **pub.del *key1 key2 key3* ...**

where the items key1, key2, etc. are a set of keys that will find the paper, as with the ***lookbib*** command. That is, if

logo% **lookbib Aho hirschberg ullman**

will find a paper,

logo% **pub.del aho hirschberg ullman**

deletes it. Note that upper and lower case are equivalent in keys. The ***pub.del*** command displays the entry being deleted. It also gives the name of a ***bibzzz*** file on which the deletion command is stored. The actual deletion is not done until the changes, additions, and so on are processed, as with the ***pub.chg*** command. If, after seeing the item to be deleted, you change your mind about throwing it away, delete the ***bibzzz*** file and the delete request disappears. Again, if the list of keys does not uniquely identify one paper, an error message is given.

Remember that the default versions of the commands described here edit a public data base. Do not delete items unless you are sure deletion is proper; usually this means that there are duplicate entries for the same paper. Otherwise, view requests for deletion with skepticism; even if one person has no need for a particular item in the data base, someone else may want it there.

If an item is correct, but should not appear in the "List of Publications" as normally produced, add the line

%K DNL

to the item. This preserves the item intact, but implies "Do Not List" to the to the commands that print publication lists. The DNL line is normally used for some technical reports, minor memoranda, or other low-grade publications.

6.5.2.5. Updating and Reindexing

When you have completed a session of changes, you should type:

```
logo% pub.run file1 file2 ...
```

where the names *file1, ...* are the new files of additions you have prepared. You need not list the *bibzzz* files representing changes and deletions; they are processed automatically. All of the new items are edited into the standard public data base, and then a new index is made. This process takes about 15 minutes; during this time, searches of the data base will be slower.

Normally, you should execute *pub.run* just before you logoff after performing some edit requests. However, if you don't, the various change request files remain in your directory until you finally do execute *pub.run*. When the changes are processed, the *bibzzz* files are deleted. Do no wait too long before processing changes, however, to avoid conflicts with someone else who wishes to change the same file. If executing *pub.run* produces the message "File bibxxx too old" it means that someone else has been editing the same file between the time you prepared your changes, and the time you typed *pub.run*. You must delete such old change files and re-enter them.

Note that although *pub.run* discards the *bibzzz* files after processing them, your files of additions are left around even after *pub.run* is finished. If they were typed in only for purposes of updating the data base, you may delete them after they have been processed by *pub.run*.

Consider the following example.

Suppose, for example, that you wish to

1. Add to the data base the memos "The Dilogarithm Function of a Real Argument" by R. Morris, and "UNIX Software Distribution by Communication Link," by M. E. Lesk and A. S. Cohen;
2. Delete from the data base the item "Cheap Typesetters", by M. E. Lesk, SIGLASH Newsletter, 1973; and
3. Change "J. Assoc. Comp. Mach." to "Jour. ACM" in the citation for Aho, Hirschberg, and Ullman shown above.

The procedure is to make a file containing the additions, here called *new.1*, in the normal way using an editor. In the script shown below, you would type what is in **boldface**.

```
logo% ed new.1
?
a
%T The Dilogarithm Function of a Real Argument
%A Robert Morris
%D 1978

%T UNIX Software Distribution by Communication Link
%A M. E. Lesk
%A A. S. Cohen
%D 1978
w new.1
199
q
logo%
```

Next you would specify the deletion with the *pub.del* command and see the following response:

```
logo% pub.cel lesk cheap typesetters siglash

Will delete: (file bib176)

%T Cheap Typesetters
%A M. E. Lesk
%J ACM SIGLASH Newsletter
%V 6
%N 4
%P 14-16
%D October 1978
```

You would then extract the Aho, Hirschberg and Ullman paper. The dialogue involved is shown below. First run *pub.chg* to extract the paper; it responds by displaying the citation and informing you that it was placed on file *bib123*. That file is then edited.

```
logo% pub.chg aho hirschberg ullman
%Extracting as file bib123
%T Bounds on th Complexity of the Maximal
Common Subsequence Problem
%A A. V.Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. Assoc. Comp. Mach.
%V 23
%N 1
%P 1-12
%M TM 75-1271-7
%D Jan. 1976
```

```
logo% ed bib123
312
/Assoc/s/J/Jour/p
%J Jour. Assoc. Comp. Mach.
s/Assoc.*/ACM/p
%J Jour. ACM
1,$p
%# /usr/dict/papers/p76 233 245 change
%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. Jour. ACM
%V 23
%N 1
%P 1-12
%M TM 75-1271-7
%D Jan. 1976

w
292
q
logo%
```

Finally, execute *pub.run*, making sure to remember that you have prepared a new file *new.1*:

```
logo% pub.run new.1
```

and about fifteen minutes later the new index is complete with all the changes.

6.5.3. Printing a Publication List

There are two commands for printing a publication list, depending on whether you want to print one person's list, or the list of many people. To print a list for one person, use the *pub.indiv* command:

```
logo% pub.indiv M Lesk
```

This runs off the list for M. Lesk and puts it in file *output*. Note that no '.' is given after the initial. If only one initial does not make it clear, use two initials. Similarly, to get the list for group of people, say

```
logo% pub.org xxx
```

which prints all the publications of the members of organization *zzz*, taking the names for the list in the file */usr/dict/papers/centlist/zzz*. Run this command in the background as it takes perhaps 15 minutes. Two options are available with these commands:

```
logo% pub.indiv -p M Lesk
```

displays only the papers, leaving out unpublished notes, patents, and so on. Also

```
logo% pub.indiv -t M Lesk | gcat
```

prints a typeset copy, instead of a computer printer copy. In this case it has been directed to an alternate typesetter with the *gcat* command. You can use these options together, and with the *pub.org* command as well. For example, to print only the papers for all of organization zzz and typeset them, type

```
logo% pub.center -t -p zzz | gcat &
```

These publication lists are printed double column with a citation style taken from a set of publication list macros; you can change the macros, of course, to adjust the format of the lists.

Table of Contents

List of Tables

Chapter 7

Formatting Documents with the –me Macros

This chapter[1] describes the *-me* macro package text processing facility. The first part of each section presents the material in user's guide format and the second part lists the macro requests for quick reference. The chapter contents include descriptions of the basic requests, displays, annotations, such as footnotes, and how to use *-me* with *nroff* and *troff*.

We assume that you are somewhat familiar with *nroff* and *troff* and that you know something about breaks, fonts, point sizes, the use and definition of number registers and strings, and scaling factors for ens, points, vertical line spaces (v's), etc. If you are a newcomer, try out the basic features as you read along.

All request names in *-me* follow a naming convention. You may define number registers, strings, and macros, provided that you use single-character, upper-case names or double character names consisting of letters and digits with at least one upper-case letter. Do not use special characters in the names you define. The word *argument* in this chapter means a word or number which appears on the same line as a request and which modifies the meaning of that request. Default parameter values are given in brackets. For example, the request

```
.sp
```

spaces one line, and

```
.sp 4
```

spaces four lines. The number '4' is an *argument* to the '.sp' request; it modifies '.sp' to produce four lines instead of one. Spaces separate arguments from the request and from each other.

7.1. Using –me

When you have your raw text ready, run the *nroff* formatter with the *-me* option to send the output to the standard output, your workstation screen. Type:

logo% **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 you omit the **–T** flag, a 'lowest common denominator' terminal is assumed; this is good for previewing output on most terminals.

For easier viewing, pipe the output to *more* or redirect it to another file.

For formatting on the phototypesetter with *troff* (or your installation's equivalent), use:

logo% **troff –me** *file*

[1] The material in this chapter is derived from *Writing Papers with 'nroff' Using '-me'*, E.P. Allman and *'-me' Reference Manual*, E.P. Allman, University of California, Berkeley.

7.2. Basic –me Requests

The following sections provide descriptions and examples of the basic *–me* requests.

7.2.1. Paragraphs

The *–me* package has requests for formatting standard, left block, and indented paragraphs.

7.2.1.1. Standard Paragraph — '.pp'

Begin standard paragraphs 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

Now is the time for all good men to come to the aid of their party. Four score and seven years ago,...

that is, a blank line followed by an indented first line.

Do not begin the sentences of a paragraph with a space, since blank lines and lines beginning with spaces cause a break. For example, if you type:

```
.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 is:

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 begins with a space character.

Because the first call to one of the paragraph macros defined in a section or the '.H' macro (see *Section Headings*) *initializes* the macro processor, do not use any of the following requests: '.sc', '.lo', '.th', or '.ac'. Also, avoid changing parameters, notably page length and header and footer margins, which have a global effect on the format of the page.

7.2.1.2. Left Block Paragraphs — '.lp'

A formatted paragraph can start with a blank line and with the first line indented. You can get left-justified block-style paragraphs as shown throughout this manual by using '.lp' (left paragraph) instead of '.pp'.

7.2.1.3. Indented Paragraphs — '.ip' and '.np'

Sometimes you want to use paragraphs that have the *body* indented, and the first line exdented, that is, the opposite of indented, with a label. Use the '.ip' request for this. A word specified on the same line as '.ip' is printed in the margin, and the body is lined up at a specified position. 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 an '.ip' request, 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', type:

```
.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' begins a new line after the label. For example, the input:

```
.ip longlabel
This paragraph has 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.
```

produces:

longlabel
This paragraph has 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.

You can change the size of the label by using a second argument which is the size of the label. For example, you can produce the above example correctly by saying:

```
.ip longlabel 10
```

which will make the paragraph indent 10 spaces for this paragraph only. For example:

longlabel

This paragraph has 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.

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'.

If you use '.ip' without an argument, no hanging tag is 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 does not have a 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 does not have a 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 '.H' 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.
.lp
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.

7.2.1.4. Paragraph Reference

.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 **\(nps** 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($1** [0] plus **\n(po** [0] unless the paragraph is inside a display (see '.ba' in *Miscellaneous Requests*). 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 is (such as with '.lp'). The title *T* is exdented. 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' starts a new line.

.np An '.ip' variant that numbers paragraphs. Numbering is reset after an '.lp', '.pp', or '.H'. The current paragraph number is in **\n$p**.

7.3. Headers and Footers — '.he' and '.fo'

You can put arbitrary headers and footers 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. The first character of *title* (whatever it may be) is used as a delimiter to separate these three parts. You can use any character but avoid the backslash and double quote marks. The percent sign is replaced by the current page number whenever it is 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.

If there are two blanks adjacent anywhere in the title or more than eight blanks total, you must enclose three-part titles in single quotes.

Headers and footers are set in font **\n(tf** [3] and size **\n(tp** [10p]. Each of the definitions applies as of the *next* page.

Three number registers control the spacing of headers and footers. **\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). You can also specify the space between the top of the page and the header, the header and the first line of text, the bottom of the text and the footer, and the footer and the bottom of the page with the macros '.m1', '.m2', '.m3', and '.m4'.

7.3.1. Headers and Footers Reference

.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. 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 headers, such as, multi-line, 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 processing the header, initial saved floating keeps, etc.; in other words, this macro is called immediately before printing text on a page. Used for column headings and the like.

7.3.2. Double Spacing — '.ls 2'

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'.

7.3.3. Page Layout

You can change the way the printed copy looks, sometimes called the ***layout*** of the output page with the following requests. Most of these requests adjust the placing of 'white space' (blank lines or spaces). In these explanations, replace characters in italics with values you wish to use; bold characters represent characters which you should actually type.

Use '.bp' (break page) to start a new page.

The request '.sp *N*' leaves *N* lines of blank space. You can omit *N* to skip a single line or you can use 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*' (indent) request changes the amount of white space on the left of the page. 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.

You can center text lines with the '.ce' (center) 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, to 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' (break).

7.3.4. Underlining — '.ul'

Use the '.ul' (underline) request to underline text. The '.ul' request operates on the next input line when it is processed. You can underline multiple lines by stating a count of *input* lines to underline, followed by those lines, the same as with the '.ce' request. For example, the input:

```
.ul 2
The quick brown fox
jumped over the lazy dog.
```

underlines those words in *nroff*. In *troff* they are italicized.

7.3.5. Displays

Use displays to set off sections of text from the body of the paper. Major quotes, tables, and figures are types of displays, as are all the examples used in this manual. All displays except centered text blocks are single spaced.

7.3.5.1. Major Quotes — '.(q' and '.)q'

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. Use '.(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,...

7.3.5.2. Lists — '.(l' and '.)l'

A *list* is an indented, single-spaced, unfilled display. You should use lists when the material to be printed should not be filled and justified like normal text. This is useful for columns of figures, for example. Surround the list text 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
```

to produce:

Alternatives to avoid deadlock are:

Lock in a specified order
Detect deadlock and back out one process
Lock all resources needed before proceeding

7.3.5.3. Keeps — '.(b' and '.)b', '.(z' and '.)z'

A *keep* is a display of lines which are kept on a single page if possible. Keeps are useful for printing diagrams, for example. 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 enough 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 a *floating keep.*

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'. An example of a floating keep is:

```
.(z
.hl
Text of keep to be floated.
.sp
.ce
Figure 1. Example of a Floating Keep.
.hl
.)z
```

The '.hl' request draws a horizontal line so that the figure stands out from the text.

7.4. Fancy 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 produced by '.(l' is 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 formatted 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, while floating keeps are normally left justified. To get a left-justified list, type '.(l L'. To center a list line-for-line, type '.(l C'. For example, to get a filled, left-justified list, use:

```
.(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 you may 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' keep argument.

Centered blocks are *not* keeps, and you may use them 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

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 centers the centered block within the entire line rather than within the line minus the indent. Also, you must nest the center requests *inside* the keep requests.

7.4.1. Display Reference

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** [4n]; if it is **M**, the list is indented to the left margin; if it is **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 it is **C**, the list is centered on a line-by-line basis. The list is set in font **\n(df** [0]. You must use a matching '.)l' to end the list. This macro is almost like '.DS' except that no attempt is made to keep the display on one page.

.)l — End list.

.(q — Begin major quote. The lines are single-spaced, filled, moved in from the main body of 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**, that is, one point smaller than the 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** [4n] 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.

7.4.2. 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 explicitly called for.

7.4.3. Footnotes — '.(f' and '.)f'

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:

```
.(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]

Make sure that the footnote appears ***inside*** the quote, so that the footnote will appear on the same page as the quote.

---

[1] [1]Like this.

[1] [2]James R. Ware, *The Best of Confucius*, Halcyon House, 1950. Page 77.

7.4.4. Delayed Text

Delayed text is very similar to a footnote except that it is printed when explicitly called for. Use this feature to put a list of references 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 refer to them with special characters* rather than numbers.

7.4.5. Indexes — '.(x' '.)x' and '.xp'

An *index* resembles delayed text, in that it is saved until called for. It is actually more like a table of contents, since the entries are not sorted alphabetically. 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'. An argument to the '.)x' indicates 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:

```
.(x
Sealing wax
.)x 9
.(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 9
.xp
```

generates:

Sealing wax 9

Cabbages and kings

< etc. >

[1] *Such as an asterisk.

The '.(x' request may have a single character argument, specifying the *name* of the index; the normal index is **x**. Thus, you can maintain several *indicies* simultaneously, such as a list of tables and a table of contents.

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); you may have to rearrange the pages after printing.

7.4.6. Annotations Reference

.(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. You might use this 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-columned 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 generate the footnote separator. You may redefine this macro to give other size lines or other types of separators. It currently draws a 1.5-inch line.

.(x *x* — Begin index entry. Index entries are saved in the index *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 formated in the font, size, and so forth in effect at the time it is printed, rather than at the time it is collected.

7.5. Fancy 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 those used in this manual, and multicolumn output.

7.5.1. Section Headings — '.sh' and '.uh'

You can automatically generate section numbers, using the '.sh' request. You must tell '.sh' the *depth* of the section number and a section title. The depth specifies how many numbers separated by decimal points are to appear in the section number. For example, the section

number '4.2.5' has a depth of three.

Section numbers are incremented if you add a number. Hence, you increase the depth, and 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 beginning section number 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 be numbered relative to this number.

There are more complex features which indent each section proportionally to the depth of the section. For example, if you type:

```
.nr si Nz
```

each section will be indented by an amount *N*. *N* must have a scaling factor attached, that is, it must be of the form *Nz*, where *z* is a character telling what units *N* is in. Common values for *z* 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
```

The request indents sections by one-half inch per level of depth in the section number. As another example, consider:

```
.nr si 3n
```

which gives three spaces of indent per section depth.

You can produce section headers without automatically generated numbers using:

```
.uh "Title"
```

which will do a section heading, but will not put a number on the section.

7.5.1.1. Section Heading Reference

.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'in *Miscellaneous Requests*). 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. A '.sh' insures that there is enough room to print the section head plus the beginning of a paragraph, which is about 3 lines total. If you specify *a* through *f*, 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 (_), the section depth and numbering is reset, but the base indent is not reset and nothing is printed. 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. Be careful if you redefine this macro, as it is quite complex and subtle.

.$0 *T B N* Callled automatically after every call to '.$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 – .$6 Traps called just before printing that depth section. May be defined to give variable spacing before sections. These macros are called from '.$p', so if you redefine that macro you may lose this feature.

7.5.2. Parts of the Standard 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
A BENCHMARK FOR THE NEW SYSTEM
.sp
by
.sp
J. P. Hacker
.)l
.bp
```

The request '.th' sets up the environment of the *nroff* processor to do a thesis. It defines the correct headers, footers, a page number in the upper right-hand corner only, sets the margins correctly, and double spaces.

Use the '.+ c *T*' request 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"
```

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 standard PhD thesis.

If the title parameter *T* is omitted from the '.+ c' request, the result is a chapter with no heading. You can also use this at the beginning of a paper.

Although papers traditionally have the abstract, table of contents, and so forth at the front, 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. At the end of the paper, give the '.+ + P' request, which begins the preliminary part of the paper. After using 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. You may use '.+ c' repeatedly to begin different parts of the front material for example, the abstract, the table of contents, acknowledgments, list of illustrations, and so on. You may also use the request '.+ + B' to begin the bibliographic section at the end of the paper. For example, the paper might appear as outlined below. (In this figure, comments begin with the sequence ' \"'.)

```
.th                          \" set for thesis mode
.fo "DRAFT"                  \" define footer for each page
.tp                          \" begin title page
.(l C                        \" center a large block
A BENCHMARK FOR THE NEW SYSTEM
.sp
by
.sp
J.P. Hacker
.)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
```

Outline of a Sample Paper

7.5.2.1. Standard Paper Reference

.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. 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. Use '.+ + ' and '.+ c' with it. This macro must be stated before initialization, that is, before the first call of a

paragraph macro or '.H'.

.++ *m H*

This request defines the section of the paper which you are typing. The section type is defined by *m*: 'C' means that you are entering the chapter portion of the paper, 'A' means that you 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 you are entering the abstract (numbered independently from 1 in Arabic numerals), and 'B' means that you are entering the bibliographic portion at the end of the paper. You can also use the variants 'RC' and 'RA', 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.%'**. Precede each section (chapter, appendix, etc.) by the '.+ c' request. When using *troff*, it is easier to put the front material at the end of the paper, so that the table of contents can be collected and generated; you can then physically move this material 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.

.$c *T* Print chapter number (from **\n(ch)** and *T*. You can redefine this macro to your liking. It is defined by default to be acceptable for a standard PhD thesis. This macro calls '$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 photo-ready papers as used by the Associattion for Computing Machines (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 for printing papers for ACM conferences. Note that this macro will not work correctly in *troff*, since it sets the page length wider than the physical width of the phototypesetter roll.

7.5.3. Two-Column Output — '.2c'

You can get two column output automatically by using the request '.2c'. This produces everything after it 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'.

7.5.3.1. Columned Output Reference

.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] columned 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.

7.5.4. Defining Macros — '.de'

A *macro* is a collection of requests and text which you may use 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, type:

```
.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 command names in *–me*, always use upper-case letters as names. Avoid the names 'TS', 'TH', 'TE', 'EQ', and 'EN'.

7.5.5. 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 use something like:

```
.(s
.(c
Text of figure
.)c
.ce
Figure 5.
\!.(x f
\!Figure 5
\!.)x
.)s
```

which will give you a figure with a label and an entry in the index 'f', presumably a list of figures index. Because the index entry is read and interpreted when the keep is read, and not when it is printed, you have is to use the magic string ' \!' at the beginning of all the lines dealing with the index. Otherwise, the page number in the index is likely to be wrong. This defers index processing until the figure is generated, and guarantees that the page number in the index is correct. The same comments apply to blocks with '.(b' and '.)b'.

7.6. Using 'troff' for Phototypsetting

You can prepare documents for either displaying on a workstation or for phototypesetting using the *troff* formatting program.

7.6.1. 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 for use with the *eqn* and *neqn* mathematical equation processors. 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, 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*.

There are also several *pseudo-fonts* available. For example, the input:

```
.u underlined
```

generates

underlined

and

```
.bx "words in a box"
```

produces

words in a box

You can also get bold italics with

```
.bi "bold italics"
```

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 give the requests:

```
.bi "some bold italics"
and
.bx "words in a box"
```

in the middle of a line, *troff* would overwrite the first and the box lines on the second would be poorly drawn.

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'; you can join input lines together without a space between them. For example, the input:

```
.u under \c
.i italics
```

generates under*italics* , but if you type:

```
.u under
.i italics
```

the result is under *italics* as two words.

7.6.2. Point Sizes — '.sz'

The phototypesetter supports different sizes of type, measured in points. The default point size is 10 points for most text and eight points for footnotes. To change the point size, type:

.sz +*N*

where *N* is the size wanted in points. The 'vertical spacing,' that is, the distance between the bottom of most letters (the *baseline*) and the adjacent line is set to be proportional to the type size.

Note: Changing point sizes on the phototypesetter is a slow mechanical operation. Consider size changes carefully.

7.6.2.1. Fonts and Sizes Reference

.sz +*P* The point size is set to *P* [10p], and the line spacing is set proportionally. The ratio of line spacing to point size is stored in **\n($r**. The ratio used internally by displays and annotations is stored in **\n($R**, although '.sz' does not use this.

.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. Underlines in *nroff*.

.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, which includes being hyphenated, so in other words, it is only safe in nofill mode.

.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, which includes being hyphenated, so in other words, it is only safe in nofill mode.

.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, which includes being hyphenated, so in other words, it is only safe in nofill mode.

7.6.3. Quotes — '*(lq' and '*(rq'

It looks better to use pairs of grave and acute accents to generate double quotes, rather than the double quote character (' " ') on a phototypesetter. For example, compare "quote" to "quote". In order to make quotes compatible between the typesetter and the workstation or a terminal, 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"
```

which generates "quoted text". Notice that you must surround the material to be quoted with double quote marks if it is more than one word.

7.7. Adjusting Macro Parameters

You may adjust a number of macro parameters. You may set fonts to a font number only. In *nroff* font 8 is underlined, and is set in bold font in *troff* (although font 3, bold in *troff*, is not underlined in *nroff*). 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.

You may use registers and strings of the form '$ *x*' in expressions but you should not change them. 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.

On daisy wheel type printers in twelve pitch, you can use the '-rx1' flag to make lines default to one eighth inch, which is 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.

7.8. Roff Support

.ix +*N* Indent, no break. Equivalent to '' in *N*.

.bl *N* Leave *N* contiguous white spaces, 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'.

.nl 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. Use this 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 generated immediately if there is room, and will otherwise be generated 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 generated.

7.9. Preprocessor Support

.EQ *m T* Begin equation. The equation is centered if *m* is 'C' or omitted, indented \n(bi [4n] 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 the *Typesetting Mathematics with 'eqn'* chapter in this manual.

.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 the *Formatting Tables with 'tbl'* chapter in this manual.

.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 '.(z' and '.)z'.

7.10. 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. For example, to produce a superscript you can type **x*[2*]**, which will produce x^2.
*]	Unsuperscript. Inverse of ***[**.
*<	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 16, 1983. Other forms of the date can be used by using **\n(dy** (the day of the month; for example, 16), ***(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 marks; double quote in *nroff*.
*–	An em dash in *troff*; two hyphens in *nroff*.

7.11. Miscellaneous Requests

.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 '.H' request performs a '.ba' request if **\n(si** [0] is not zero, and sets the base indent to **\n(si*\n($0**.
.ll +*N*	Set the line length to *N* [6.0i]. This differs from '.ll' because it only affects the current environment.
.ll +*N*	Set line length in all environments to *N* [6.0i]. Do not use this after output has begun, and particularly not in two-columned output. The current line length is stored in **\n($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.
.lo	This macro loads another set of macros in */usr/lib/me/local.me* which is 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.

7.12. Special Characters and Diacritical Marks — '.sc'

There are a number of special characters and diacritical marks, such as accents, available with *-me*. To use these characters, you must call the macro '.sc' to define the characters before using them.

.sc Define special characters and diacritical marks. You must state this macro before initialization.

The special characters available are listed below.

Table 7-1: Special Characters and Diacritical Marks

Name	Usage	Example	
Acute accent	*´	a*´	á
Grave accent	*`	e*`	è
Umlaut	*:	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		∀

7.13. '–me' Request Summary

Table 7-2: –me Request Summary

Request	Initial Value	Cause Break	Explanation
.(c	–	yes	Begin centered block.
.(d	–	no	Begin delayed text.
.(f	–	no	Begin footnote.
.(l	–	yes	Begin list.
.(q	–	yes	Begin major quote.
.(x *z*	–	no	Begin indexed item in index *z*.
.(z	–	no	Begin floating keep.
.)c	–	yes	End centered block.
.)d	–	yes	End delayed text.
.)f	–	yes	End footnote.
.)l	–	yes	End list.
.)q	–	yes	End major quote.
.)x	–	yes	End index item.
.)z	–	yes	End floating keep.
.++ *m H*	–	no	Define paper section. *m* defines the part of the paper and can be C (chapter), A (appendix), P (preliminary, for example, abstract, table of contents, etc.), B (bibliography), RC (chapters renumbered from page one each chapter), or RA (appendix renumbered from page one).
.+c *T*	–	yes	Begin chapter (or appendix, etc., as set by '.++'). *T* is the chapter title.
.1c	1	yes	One column format on a new page.
.2c	1	yes	Two column format.
.EN	–	yes	Space after equation produced by *eqn* or *neqn*.
.EQ *x y*	–	yes	Precede equation; break out and add space. Equation number is *y*. The optional argument *x* may be *I* to indent equation (default), L to left-adjust the equation, or C to center the equation.
.TE	–	yes	End table.
.TH	–	yes	End heading section of table.
.TS *x*	–	yes	Begin table; if *x* is H, table has repeated heading.
.ac *A N*	–	no	Set up for ACM style output. *A* is the Author's name(s), *N* is the total number of pages. Must be given before the first initialization.

Request	Initial Value	Cause Break	Explanation
.b *x*	no	yes	Print *x* in boldface; if no argument switch to boldface.
.ba +*n*	0	yes	Augments the base indent by *n*. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column.
.bi *x*	no	no	Print *x* in bold italics (nofill only).
.bx *x*	no	no	Print *x* in a box (nofill only).
.ef '*x*'*y*'*z*'	''''	no	Set even footer to x y z.
.eh '*x*'*y*'*z*'	''''	no	Set even header to x y z.
.fo '*x*'*y*'*z*'	''''	no	Set footer to x y z.
.he '*x*'*y*'*z*'	''''	no	Set header to x y z.
.hl	–	yes	Draw a horizontal line.
.hx	–	no	Suppress headers and footers on next page.
.i *x*	no	no	Italicize *x*; if *x* is missing, italic text follows.
.ip *x y*	no	yes	Start indented paragraph, which hanging tag *x*. Indentation is *y* ens (default 5).
.lp	yes	yes	Start left-block paragraph.
.lo	–	no	Read in a file of local macros of the form '.*x*'. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of '*x*'*y*'*z*'	''''	no	Set odd footer to x y z.
.oh '*x*'*y*'*z*'	''''	no	Set odd header to x y z.
.pd	–	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.
.r	yes	no	Roman text follows.
.re	–	no	Reset tabs to default values.
.sc	–	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh *n x*	–	yes	Section head follows, font automatically bold. *n* is level of section, *x* is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.
.sz + *n*	10p	no	Increase the point size by *n* points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.
.u *x*	–	no	Underline argument (even in *troff*) (nofill only).
.uh	–	yes	Like '.sh' but unnumbered.
.xp *x*	–	no	Print index *x*.

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:

Please list typographical errors by page number and actual text of the error.

Technical Errors:

Please list errors of fact by page number and actual text of the error.

Content:

Did this guide meet your needs? If not, please indicate what you think should be added or deleted in order to do so. Please comment on any material which you feel should be present but is not. Is there material which is in other manuals, but would be more convenient if it were in this manual?

Layout and Style:

Did you find the organization of this guide useful? If not, how would you rearrange things? Do you find the style of this manual pleasing or irritating? What would you like to see different?

Part Number 800-1114-01
Revision: C of 7th January 1984
For: Sun System Release 1.1

FORTRAN and Pascal

for the

Sun Workstation

Sun Microsystems, Inc.,
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

FORTRAN and Pascal

Table of Contents

† UNIX is a trademark of Bell Laboratories.

Fortran Programmer's Guide

for the

Sun Workstation

Credits and Acknowledgements

This manual is based on Bell Laboratories document entitled *A Portable Fortran 77 Compiler*, by S. I. Feldman and P. J. Weinberger, dated 1 August 1978. Material on the I/O Library was derived from the paper entitled *Introduction to the f77 I/O Library*, by David L. Wasley, University of California, Berkeley, California 94720. Further work was done at Sun Microsystems.

Trademarks

Revision History

Revision	Date	Comments
A	15 July 1983	First release of this Programmer's Guide.
B	1 November 1983	Incorporates corrections.
C	7 January 1984	Reorganized and some extra material added.

Fortran Programmer's Guide

Table of Contents

Fortran Programmer's Guide

Table of Contents

List of Tables

Chapter 1

Introduction

The language known as FORTRAN-77 became an official American National Standard in 1978. This *Programmer's Guide* provides a description of using FORTRAN-77 in the environment of the Sun Workstation. The Sun *f77* compiler generates code compatible with calling sequences produced by the C compiler. This guide is not a language reference manual — at this time the reader is directed to one of the many books on the market, or to the ANSI standard as a last resort.

All examples in this user's guide assume that you are working on a host called **fortran**. What the user types is shown in **bold face text like this** and the system's replies are shown in the standard Roman font like the rest of this sentence.

The next section in this guide provides a very brief introduction to using the *f77* compiler on the Sun Workstation. It is assumed that you are familiar with at least the basic aspects of the UNIX† system, its file system, and some of the commands you need to find your way around. In addition to *f77*, there are some other tools you may wish to know about, and they are summarized here.

Text Editing

The major text editor for source programs is the *vi* (vee-eye) visual editor. *Vi* has considerable power and is specially directed at program source text. For more information, see *Editing and Text Processing*.

Fortran Tools

Fpr

is a FORTRAN 'output filter' for printing files which have FORTRAN carriage-control characters in columne one. As noted in the appendix describing deviations from the ANSI standard, the UNIX implementation on the Sun system does not know about carriage control since there are no explicit printer files. Thus you use *fpr* when it is vitally necessary to have FORTRAN files with carriage-control in them.

Ratfor

is 'Rational Fortran' — a preprocessor intended to add some reasonable control structures to Fortran. Ratfor was written in the days of FORTRAN-66 and is not so useful for FORTRAN-77 which has some control structures.

Debug Aids

There are two main debug tools available on the Sun system:

† UNIX is a trademark of Bell Laboratories.

Dbx
is a symbolic debugger that understands FORTRAN-77 programs.

Adb
is the 'old' debugger — it is not as easy to use as *dbx*.

1.1. Using the FORTRAN-77 Compiler on the Sun Workstation

Source code for FORTRAN-77 programs should be in files whose names end in *.f*. *F77* compiles the source code in such *.f* files and generates object code in files whose names end in *.o*. Finally, *f77* calls up the UNIX loader to create an executable file whose name is *a.out*. *F77* also understands other filename extensions (such as *.r* for Ratfor files — these topics are discussed later on).

As an example, here is just about the simplest FORTRAN-77 program, which just displays a message on the workstation's screen:

```
tutorial% cat greetings.f
      program greetings

      print *, 'Real programmers hack Fortran!'
      end
tutorial%
```

You compile *greetings* using the *f77* command like this:

```
tutorial% f77 greetings.f
greetings.f:
   MAIN greetings:
tutorial%
```

Note that *f77* displays messages indicatring the state of the compilation while it proceeds. The results of the compilation end up on a file called *a.out* and you can then run that program by typing the *a.out* command line:

```
tutorial% a.out
  Real programmers hack Fortran!
tutorial%
```

It might be inconvenient to have the results of every different FORTRAN-77 compilation ending up on a file called *a.out*. One way to change this situation is to just change the name of the file after the compilation has completed, using the *mv* command, but the better way is to tell the *f77* compiler to place the executable file on a file of the same name as the source, minus the *.f* suffix:

```
tutorial% f77 -o greetings greetings.f
greetings.f:
   MAIN greetings:
tutorial%
```

and then you run the program just by typing:

```
tutorial% greetings
  Real programmers hack Fortran!
tutorial%
```

This concludes the basic introduction to the *f77* compiler. The remainder of this chapter discusses the kinds of files that *f77* understands, the options that you may type on the *f77* command line, and covers other topics such as how to use the loader and the *dbx* debug utility.

1.2. Source Files that F77 Understands

F77 is a general-purpose 'driver' command for compiling and loading FORTRAN-77 and FORTRAN -related files. As mentioned above, FORTRAN-77 source code is contained in files having an *.f* suffix. In addition to the *.f* suffix filenames, *f77* understands these filename conventions:

Table 1-1: Filename Suffixes that F77 Understands

Suffix	Language	Action
.f	FORTRAN-77	FORTRAN-77 source programs — are compiled, and each object program is left on the file in the current directory whose name is that of the source with *.o* substituted for *.f*.
.F	FORTRAN-77	FORTRAN-77 source programs which are processed by the C preprocessor before being compiled by *f77*.
.c	C	C source files are compiled by the C compiler. The *f77* and *cc* commands generate slightly different loading sequences, since FORTRAN-77 programs need a few extra libraries and a different startup routine than do C programs.
.r	Ratfor	Ratfor source files are processed by the Ratfor preprocessor before being compiled by *f77*.
.s	Assembler	Assembler source files are processed by the assembler.
.o	Object Files	Object files are passed through to the loader.

1.3. Options to the F77 Command

The list below is the complete list of options that *f77* understands.

-c Suppress loading and produce *.o* files for each source file.

-g Produce additional symbol table information for *dbx*(1). Also pass the **-lg** flag to *ld*(1).

-w Suppress all warning messages. If the option is **-w66**, only FORTRAN-66 compatibility warnings are suppressed.

-D*name*=*def*

–D*name*
Define the *name* to the C preprocessor, as if by '#define'. If no definition is given, the name is defined as "1". (*.F* suffix files only).

–I*dir*
'#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in –I options, then in directories on a standard list. (*.F* suffix files only).

–p Prepare object files for profiling, see *prof*(1).

–pg
Produce counting code in the manner of **–p**, but invoke a run-time recording mechanism that keeps more extensive statistics and produces a *gmon.out* file at normal termination. An execution profile can then be generated by use of *gprof*(1).

–O Optimize the object-code.

–S Compile the named programs, and leave the assembler-language output on corresponding files suffixed *.s* (no *.o* file is created).

–o *output*
Name the final output file *output* instead of *a.out*.

–go
Produce additional symbol table information in an older format used by the *sdb* debugger and can still be used by the *adb*(1S) debugger, which has not yet been converted to the new format of **–g**. Also pass the **–lg** flag to *ld*(1).

–fsky
Generate code which assumes the presence of a SKY floating-point processor board. Programs compiled with this option can only be run in systems that have a SKY board installed. Programs compiled without the **–fsky** option will use the SKY board, but won't run as fast as they would if the **–fsky** option were used. If any part of a program is compiled using the **–fsky** option, you must also use this option when loading with the *f77* command, since a different set of startup routines is used.

The following options are peculiar to *f77*:

–i2 On machines which support short integers, make the default integer constants and variables short. (**–i4** is the standard value of this option). All logical quantities will be short.

–m Apply the M4 preprocessor to each *.r* file before transforming it with the Ratfor preprocessor.

–onetrip
Compile DO loops that are performed at least once if reached. FORTRAN-77 DO loops are not performed at all if the upper limit is smaller than the lower limit.

–u Make the default type of a variable 'undefined' rather than using the default FORTRAN rules.

–v Print the version number of the compiler, and the name of each pass as it executes.

–C Compile code to check that subscripts are within declared array bounds.

–F Apply the Ratfor preprocessor to relevant files, put the result in the file with the suffix changed to *.f*, but do not compile.

–R *x*
Use the string *x* as a Ratfor option in processing *.r* files.

–N[qxscn]*nnn*

Make static tables in the compiler bigger. *F77* complains if tables overflow and suggests you apply one or more of these flags. These flags have the following meanings:

- **q** Maximum number of equivalenced variables. Default is 150.
- **x** Maximum number of external names (common block names, subroutine and function names). Default is 200.
- **s** Maximum number of statement numbers. Default is 401.
- **c** Maximum depth of nesting for control statements (for example, DO loops). Default is 20.
- **n** Maximum number of identifiers. Default is 1009.

–U Do not convert upper case letters to lower case. The default is to convert FORTRAN programs to lower case except within character string constants.

Other arguments are taken to be either loader option arguments, or F77-compatible object programs, typically produced by an earlier run, or perhaps libraries of F77-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program called *a.out*.

Other flags, all library names (arguments beginning –l), and any names not ending with one of the understood suffixes are passed to the loader.

Chapter 2

Fortran Input and Output

2.1. Structure of FORTRAN-77 Files

FORTRAN-77 requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

FORTRAN-77 I/O is based on 'records'. When a direct file is opened in a FORTRAN-77 program, the record length of the records must be given, and this is used by the FORTRAN-77 I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except FORTRAN-77 I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The FORTRAN-77 I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system writes a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect is that the single record the user thought he wrote is treated as more than one record when being read or backspaced over.

2.2. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named *fort.n*. These files need not exist, nor are they created unless their units are used without first executing an **open**. The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly **opened** for sequential I/O is initially positioned. The I/O system will position the file at the beginning. Therefore, a

write will destroy any data already in the file, but a read will work reasonably. To position a file to its end, use a 'read' loop, or the system dependent 'fseek' function. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

The *f77* I/O Library implements the ANSI X3.9 1978 FORTRAN-77 standard input and output with a few minor exceptions. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX operating system.

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of FORTRAN-77 input and output. Several enhancements and extensions to FORTRAN-77 I/O have been added. The f77 library routines use the C stdio library routines to provide efficient buffering for file I/O.

2.3. FORTRAN I/O

The requirements of the ANSI standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very 'expensive' while direct access binary I/O is usually very efficient. Because of the complexity of FORTRAN-77 I/O, some general concepts deserve clarification.

2.3.1. Types of I/O

There are three forms of I/O, namely: **formatted, unformatted,** and **list-directed**. The last is related to formatted but does not obey all the rules for formatted I/O. There are two modes of access to **external** and **internal** files: **direct** and **sequential**. The definition of a logical record depends upon the combination of I/O form and mode specified by the FORTRAN-77 I/O statement.

2.3.1.1. Direct access

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

2.3.1.2. Sequential access

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The 'newline' character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with 'newline' characters to be read or written.

2.3.1.3. List directed I/O

Logical record length for **list-directed** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents.

2.3.1.4. Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** I/O is not allowed on "internal" files.

2.3.2. I/O execution

Note that each execution of a FORTRAN-77 **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a FORTRAN-77 **formatted** I/O statement causes one or more logical records to be read or written.

A slash, '/', will terminate assignment of values to the input list during **list-directed** input and the remainder of the current input line is skipped. Text following the slash is ignored and may be used to comment the data file.

Direct access list-directed I/O is not allowed. **Unformatted internal** I/O is not allowed. Both the above will be caught by the compiler. All other flavors of I/O are allowed, although some are not part of the ANSI standard.

Any error detected during I/O processing will cause the program to abort unless alternative action has been provided specifically in the program. Any I/O statement may include an **err=** clause (and **iostat=** clause) to specify an alternative branch to be taken on errors (and return the specific error code). Read statements may include **end=** to branch on end-of-file. File position and the value of I/O list items is undefined following an error.

2.4. Implementation details

Some details of the current implementation may be useful in understanding constraints on FORTRAN-77 I/O.

2.4.1. Number of logical units

The maximum number of logical units that a program may have open at one time is the same as the UNIX system limit, currently 20.

2.4.2. Standard logical units

By default, logical units 0, 5, and 6 are opened to 'stderr', 'stdin', and 'stdout' respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to close the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell re-direction to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the open statement specifying the unit number and no file name (see below).

The standard units, 0, 5, and 6, are named internally 'stderr', 'stdin', and 'stdout' respectively. These are not actual file names and can not be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

2.4.3. Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **form = 'print'**. Control codes '0' and '1' are replaced in the output file with '\n' and '\f' respectively. The control character '+' is not implemented and, like any other character in the first position of a record written to a 'print' file, is dropped. No vertical format control is recognized for **direct formatted** output or **list directed** output. See *fpr*(1) for an alternative way of mapping Fortran carriage control to ASCII control characters.

2.4.4. The open statement

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank=** and **form=** specifiers may be redefined without affecting the current file position. Otherwise, if **status = 'scratch'** is specified, a temporary file with a name of the form 'tmp.FXXXXXX' will be opened, and, by default, will be deleted when closed or during termination of program execution. Any other **status=** specifier without an associated file name results in opening a file named 'fort.N' where N is the specified logical unit number.

It is an error to try to open an existing file with **status = 'new'** . It is an error to try to open a nonexistent file with **status = 'old'** . By default, **status = 'unknown'** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close** , **backspace** , or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

2.4.5. Format interpretation

Formats are parsed at the beginning of each execution of a formatted I/O statement. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*). On **Ew.dEe** output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if 'e' is zero (see appendix B).

On output, a real value that is truly zero will display as '0.' to distinguish it from a very small non-zero value. This occurs in **F** and **G** format conversions. This was not done for **E** and **D**

since the embedded blanks in the external datum causes problems for other input systems.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is not, it will have a different meaning.

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

2.4.6. List directed output

In formatting list directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List-directed output of **complex** values includes an appropriate comma. List-directed output distinguishes between **real** and **double precision** values and formats them differently. Output of a character string that includes '\n' is interpreted reasonably by the output system.

2.4.7. I/O errors

If I/O errors are not trapped by the user's program an appropriate error message will be written to 'stderr' before aborting. An error number will be printed in [ ] along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to UNIX errors; these are described in *intro*(2) in the Sun *System Interface Manual.* Error numbers ≥ 100 come from the I/O library, and are described further in the appendix to this writeup. For internal I/O, part of the string will be printed with '|' at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

2.5. Non-'ANSI Standard' extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

2.5.1. Format specifiers

B is an acceptable edit control specifier. It causes return to the default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the **Ew.dEe** format specifier has been extended to **D** also. The form **Ew.d.e** is allowed but is not standard. The 'e' field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the 'e' field will be filled with asterisks (*). The default value for 'e' is 2.

An additional form of tab control specification has been added. The ANSI standard forms **TRn**, **TLn**, and **Tn** are supported where *n* is a positive non-zero number. If **T** or **nT** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign (**$**). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P**, the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as **[n]R** where $2 \leq n \leq 36$. If *n* is omitted, the default decimal radix is restored.

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete. Radix and 'unsigned' specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 16R, 8I10.8 )
```

Note: Unsigned integer values greater than (2**30 - 1), i.e. any signed negative value, can not be read by FORTRAN-77 input routines. All internal values will be output correctly.

2.5.2. Print files

The ANSI standard is ambiguous regarding the definition of a 'print' file. Since UNIX has no default 'print' file, an additional **form==** specifier is now recognized in the **open** statement. Specifying **form = 'print'** implies **formatted** and enables vertical format control for that logical unit. Vertical format control is interpreted only on sequential formatted writes to a 'print' file.

The **inquire** statement will return **print** in the **form==** string variable for logical units opened as 'print' files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form==** option or the **blank==** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit==** refers to a standard input or output. Therefore, to re-define the standard output as a 'print' file, use:

```
open (unit=6, form='print')
```

2.5.3. Scratch files

A **close** statement with **status = 'keep'** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file's real name, using **inquire** , if you want to re-open it later.

2.5.4. List directed I/O

List directed read has been modified to allow input of a string not enclosed in quotes. The string must not start with a digit, and can not contain a separator (, or /) or blank (space or tab). A newline will terminate the string unless escaped with \. Any string not meeting the above restrictions must be enclosed in quotes (" or ').

Internal list-directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the 'end-of-file' is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

2.6. Running older programs

Traditional FORTRAN-77 environments usually assume carriage control on all logical units, usually interpret blank spaces on input as '0's, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

2.6.1. Traditional unit control parameters

If a program reads and writes only units 5 and 6, then including **–lI66** in the **f77** command will cause carriage control to be interpreted on output and cause blanks to be zeros on input without further modification of the program. If this is not adequate, the routine *ioinit*(3f) can be called to specify control parameters separately, including whether files should be positioned at their beginning or end upon opening.

2.6.2. Preattachment of logical units

The *ioinit* routine also can be used to attach logical units to specific files at run time. It will look for names of a user specified form in the environment and open the corresponding logical unit for **sequential formatted** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading '0'.

Ioinit should prove adequate for most programs as written. However, it is written in FORTRAN-77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by 'ar x /usr/lib/libI77.a ioinit.f'.

2.7. Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal FORTRAN-77 I/O processing and do not obey FORTRAN-77 I/O rules. See *tapeio*(3f).

This example is admittedly far-fetched, but illustrates that the error messages are sufficiently clear to allow easy determination of the problem in the expressions.

2.2.7. Type equivalence

Several diagnostics produced by the Pascal translator complain about 'non-equivalent types'. In general, Berkeley Pascal considers variables to have the same type only if they were declared with the same constructed type or with the same type identifier. Thus, the variables *x* and *y* declared as

```
var
   x: ↑ integer;
   y: ↑ integer;
```

do not have the same type. The assignment

```
x := y
```

thus produces the diagnostics:

```
Wed Nov  9 15:52 1983  typequ.p:
E 7 - Type clash: non-identical pointer types
  ... Type of expression clashed with type of variable in assignment
```

Thus it is always necessary to declare a type such as

```
type intptr = ↑ integer;
```

and use it to declare

```
var x: intptr; y: intptr;
```

Note that if we had initially declared

```
var x, y: ↑ integer;
```

then the assignment statement would have worked. The statement

```
x↑ := y↑
```

is allowed in either case. Since the parameter to a **procedure** or **function** must be declared with a type identifier rather than a constructed type, it is always necessary, in practice, to declare any type which will be used in this way.

2.2.8. Unreachable statements

Berkeley Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic. Note that a statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no diagnostic is produced for the statement:

```
if false then
   writeln('impossible!')
```

Chapter 3

Input/output

This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.

3.1. Introduction

Our first sample programs, in the *Basic UNIX Pascal* section, used the file *output*. We gave examples there of redirecting the output to a file and to the line printer using the shell. Similarly, we can read the input from a file or another program. Consider the following Pascal program which is similar to the program *cat* (1).

```
tutorial% pix -l kat.p <primes
Berkeley Pascal PI -- Version 2.13 (4/7/83)

Wed Nov  9 15:53 1983  kat.p

    1  program kat(input, output);
    2  var
    3      ch: char;
    4  begin
    5      while not eof do begin
    6          while not eoln do begin
    7              read(ch);
    8              write(ch)
    9          end;
   10          readln;
   11          writeln
   12      end
   13  end { kat }.
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229

Execution begins...
Execution terminated.
```

```
925 statements executed in 0.20 seconds cpu time.
tutorial%
```

Here we have used the shell's syntax to redirect the program input from a file in *primes* in which we had placed the output of our prime number program in the *Execution profiling* section. It is also possible to 'pipe' input to this program much as we piped input to the line printer daemon *lpr* (1) before. Thus, the same output as above would be produced by

```
tutorial% cat primes | pix -l kat.p
```

All of these examples use the shell to control the input and output from files. One very simple way to associate Pascal files with named UNIX files is to place the file name in the **program** statement. For example, suppose we have previously created the file *data*. We then use it as input to another version of a listing program.

```
tutorial% cat data
line one.
line two.
line three is the end.
tutorial% pix -l copydata.p
Berkeley Pascal PI -- Version 2.13 (4/7/83)

Wed Nov  9 15:53 1983  copydata.p

    1  program copydata(data, output);
    2  var
    3      ch: char;
    4      data: text;
    5  begin
    6      reset(data);
    7      while not eof(data) do begin
    8          while not eoln(data) do begin
    9              read(data, ch);
   10              write(ch)
   11          end;
   12          readln(data);
   13          writeln
   14      end
   15  end { copydata }.
line one.
line two.
line three is the end.
Execution begins...
Execution terminated.

134 statements executed in 0.05 seconds cpu time.
tutorial%
```

By mentioning the file *data* in the **program** statement, we have indicated that we wish it to correspond to the UNIX file *data*. Then, when we 'reset(data)', the Pascal system opens our file 'data' for reading. More sophisticated, but less portable, examples of using UNIX files will be given in the *Output buffering* and *Files, reset, and rewrite* sections below. There is a portability

ones), and a zero mantissa.

Not-a-Number (NaN)
is represented by the largest value that the exponent can assume (all ones), and a non-zero mantissa. The sign is usually ignored.

Normalized **real** and **double precision** numbers are said to contain a 'hidden' bit, providing for one more bit of precision than would normally be the case.

The largest non-infinite **double precision** number is approximately 1.797693e+308; the smallest positive normalized **double precision** number is approximately 2.225074e-308. The largest non-infinite **real** number is approximately 3.402823e+38; the smallest positive, normalized **real** number is approximately 1.175494e-38.

3.2.3. Hexadecimal Representation of Selected Numbers

Table 3-2: Hexadecimal Representation of Selected Numbers

Value	Real	Double Precision
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxx

3.2.4. Deviations from the Proposed IEEE Standard

Deviations from the proposed IEEE standard in this implementation are as follows:

- affine mapping for infinities,
- normalizing mode for denormalized numbers,
- rounds approximately to nearest - 7 or more guard bits are computed, but the 'sticky' bit is not,
- exception flags are not implemented.

3.2.5. Arithmetic Operations on Extreme Values

This subsection describes the results derived from applying the basic arithmetic operations on combinations of extreme values and ordinary values.

No traps or any other exception actions are taken.

All inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen.

In all the tables below, the abbreviations have the following meanings:

Table 3-3: Meaning of Abbreviations for Numbers

Abbreviation	Meaning
Den	Denormalized Number
Num	Normalized Number
Inf	Infinity (positive or negative)
NaN	Not a Number
Uno	Unordered

Addition and Subtraction					
Left Operand	Right Operand 0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Note 1	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Note 1: Inf + Inf = Inf; Inf - Inf = NaN

Multiplication					
Left Operand	Right Operand 0	Den	Num	Inf	NaN
0	0	0	0	NaN	NaN
Den	0	0	Num	Inf	NaN
Num	0	Num	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Division					
Left Operand	Right Operand 0	Den	Num	Inf	NaN
0	NaN	0	0	0	NaN
Den	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	NaN	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Comparison					
Left Operand	Right Operand 0	Den	Num	Inf	NaN
0	=	<	<	<	Uno
Den	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>		Uno
Nan	Uno	Uno	Uno	Uno	Uno

- NaN compared with NaN is Unordered, and also results in inequality.
- + 0 compares equal to -0.

Max					
Left Operand	Right Operand 0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Inf	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Min					
Left Operand	Right Operand 0	Den	Num	Inf	NaN
0	0	0	0	0	NaN
Den	0	Den	Den	Den	NaN
Num	0	Den	Num	Num	NaN
Inf	0	Den	Num	Inf	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Chapter 4

Inter-Procedure Interface

To be able to write C procedures that call or are called by FORTRAN-77 procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

F77 appends an underscore to the name of a common block or procedure to distinguish it from C procedures or external variables with the same user-assigned name. FORTRAN-77 library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding FORTRAN-77 and C declarations:

Table 4-1: Corresponding FORTRAN and C Declarations

FORTRAN	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

By the rules of FORTRAN-77, **integer**, **logical**, and **real** data occupy the same amount of memory.

4.3. Return Values

A function of type **integer**, **logical**, **real**, or **double precision** declared as a C function that returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . .)
struct { float r, i; } *temp;
 . . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments, namely a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . .)
char result[ ];
long int length;
 . . .
```

and could be invoked in C by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were integer-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

4.4. Argument Lists

All FORTRAN-77 arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. The string lengths are **long int** quantities passed by value. The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to that in:

```
int f_();
char s[7];
long int b[3];
 . . .
sam_(f_, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but FORTRAN-77 arrays begin at 1 by default. FORTRAN-77 arrays are stored in column-major order, C arrays are stored in row-major order.

Appendix A

Deviations from the Fortran-77 Standard

FORTRAN-77 includes almost all of FORTRAN-66 as a subset. Appendix B contains a brief description of the differences between FORTRAN-66 and FORTRAN-77.

The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

This appendix is in two major parts: the first part describes extensions to the ANSI standard that this FORTRAN compiler and run-time system inplement. The second part describes areas where this compiler and run-time system violate the ANSI standard, usually because the compiler or rune-time system cannot correctly implement the ANSI standard.

A.1. Extensions to the FORTRAN-77 Standard

In addition to implementing the language specified in the ANSI Standard, the Sun *f77* compiler implements some extensions as described in this chapter. Some of the extensions are useful additions to the language. The remaining extensions make it easier to communicate with C procedures or to permit compilation of old FORTRAN-66 programs.

A.1.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

A.1.2. Internal Files

The FORTRAN-77 standard introduces 'internal files' (memory arrays), but restricts their use to formatted sequential I/O statements. The Sun *f77* I/O system also permits internal files to be used in direct reads and writes.

A.1.3. Implicit Undefined statement

FORTRAN-66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m,** or **n**, and **real** otherwise. FORTRAN-77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, the Sun *f77* compiler has an additional **undefined** data type. The statement:

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and *f77* issues a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **–u** compiler flag is equivalent to beginning each procedure with this statement.

A.1.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures. But note that a subroutine or function may not pass its own name as a procedure parameter. To do so would require the name to appear in an **external** statement, which is prohibited by the ANSI standard. Note also that use of recursion makes FORTRAN programs non-portable.

A.1.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic**. These keywords may appear as 'types' in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence**, **data**, or **save** statements.

A.1.6. Source Input Format

The Standard expects input to *f77* to be in 72 column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next sixty-six are the body of the line. If a line of this format contains fewer than 72 characters, *f77* pads it with blanks. Characters after the seventy-second are ignored.

In order to make it easier to type FORTRAN-77 programs, this compiler also accepts input in variable length lines. An ampersand ('&') in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by *f77*. Lines containing a tab among the first six characters, or lines beginning with an ampersand, are not padded with blanks, nor does *f77* ignore characters past the 72nd character in lines of this format.

In the Standard, there are only 26 letters — FORTRAN-77 is a one-case language. Consistent with ordinary UNIX system usage, this compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **–U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. However, FORTRAN-77 reserved words are only recognized in lower case.

A.1.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**. **includes** may be nested to a reasonable depth, currently ten.

A.1.8. Binary Initialization Constants

A **logical**, **real**, or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0–7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0–9**, **a–f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

A.1.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

Table A-1: Backslash Escape Sequences

Character	Meaning
\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	\
\z	z, where z is any other character

FORTRAN-77 only has one quoting character, namely the apostrophe. This compiler and I/O system recognize both the apostrophe (') and the double-quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

A.1.10. Hollerith

FORTRAN-77 does not have the old Hollerith (**nh**) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

A.1.11. Equivalence Statements

As a very special and peculiar case, FORTRAN-66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. FORTRAN-77 does not permit this usage, since subscript lower bounds may now be different from 1. The Sun *f77* compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

A.1.12. One-Trip DO Loops

The FORTRAN-77 standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The FORTRAN-66 standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the FORTRAN-66 standard, the **–onetrip** compiler flag makes *f77* generate non-standard loops.

A.1.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
–345,.05e–3,12
```

correctly.

A.1.14. Short Integers

On machines that support halfword integers, *f77* accepts declarations of type **integer*2**. Ordinary integers follow the FORTRAN-77 rules about occupying the same space as a **real** variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**. An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **–i2** flag, all small integer constants will be of type **integer*2**. If the precision of

an integer-valued intrinsic function is not determined by the generic function rules, one is chosen that returns the prevailing length (**integer*2** when the **–i2** command flag is in effect). When the **–i2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

A.1.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the FORTRAN-77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or**, **and**, **xor**, and **not**) and for accessing the UNIX command arguments (**getarg** and **iargc**) and environment (**getenv**).

A.2. Violations of the Standard

There are only a few ways in which this implementation of FORTRAN-77 system violates the ANSI FORTRAN-77 standard:

A.2.1. Dummy Procedure Arguments

If any argument of a procedure is of type **character**, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external**. Code is correct if there are no **character** arguments.

A.2.2. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The I/O library uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by FORTRAN-77 or the operating system.

A.2.3. Carriage Control

The ANSI standard leaves as implementation-dependent which logical unit(s) are treated as 'printer' files. In this implementation, there is no printer file and thus carriage control specifiers such as '+ ' are not implemented. It would be difficult to implement these carriage-control characters correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation can not be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a FORTRAN 'output filter' before printing. See the *fpr*(1) command in the *User's Manual*.

A.2.4. Assigned Goto

The optional *list* associated with an assigned **goto** statement is not checked against the actual assigned value during execution.

A.2.5. Default files

Files created by default use of **rewind** or **endfile** statements are opened for **sequential formatted** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

A.2.6. Lower case strings

It is not clear if the ANSI standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement returns lower case strings for any alphanumeric data.

A.2.7. Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the ANSI standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, 'w', should be filled with asterisks if the exponent can not be displayed. This system fills only the exponent field in the above case since that is more informative.

A.2.8. Repeat counts for null values

Repeat counts for null values on list-directed input are not recognized correctly.

Appendix B

Differences Between Fortran-66 and Fortran-77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with FORTRAN-66.

B.1. Features Deleted from FORTRAN-66

B.1.1. Hollerith

All notions of 'Hollerith' (*n* h) as data have been officially removed, although this compiler, like almost all in the foreseeable future, still supports this anachronism.

B.1.2. Extended Range

In FORTRAN-66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a do loop, then jump back into it. Extended range has been removed in the FORTRAN-77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

B.2. Program Form

B.2.1. Blank Lines

Completely blank lines are now legal comment lines.

B.2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. This rule is not enforced by this system. The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

B.2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value.

Arguments do not retain their values between calls. The ancient trick of calling one entry point with a large number of arguments so that the procedure 'remembers' the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in this implementation, since arguments are not kept in static storage.

B.2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use. The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u-l)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

B.2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the 'alternate returns' is described in section 5.2 of this appendix.

B.3. Declarations

B.3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

There is an intrinsic function **len** that returns the actual length of a character string. Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

B.3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer**, other variables are of type **real**, unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real**, those beginning with **w, x, y,** or **z** are assumed **complex**, and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

B.3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello' )
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

B.3.4. Array Declarations

Arrays may now have as many as seven dimensions — only three were permitted in 1966. The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common**

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

B.3.5. SAVE Statement

A poorly known rule of FORTRAN-66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. The only exceptions are variables that have been defined in a **data** statement and never changed. These rules permit overlay and stack implementations for the affected variables. FORTRAN-77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **saved** in every procedure in which it is declared if the desired effect is to occur.

B.3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, 'intrinsic functions', rather than being divided into 'intrinsic' and 'basic external' functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic** Declaring it **external** (as in FORTRAN-66) passes a function other than the built-in one.

B.4. Expressions

B.4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in FORTRAN-77. Our compiler has two different quotation marks, " ' " and " " ".

B.4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash ('//'). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a '*(*)' modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

B.4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

B.4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leq n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

B.4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. The principal part of the logarithm is used. Also, multiple exponentiation is now defined:

```
a**b**c = a ** (b**c)
```

B.4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. For instance, it is permissible to combine integer and complex quantities in an expression.

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameters** and the FORTRAN operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in **common**.

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

B.5. Executable Statements

B.5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to FORTRAN. It is called a 'Block If'. A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if(. . .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif**, **else**, or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the **elseif** conditions are true, control passes to the statements following the **else** statement, if any. The **else** must follow all **elseifs** in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures. A case construct may be rendered

```
if (s .eq. 'ab') then
 . . .
else if (s .eq. 'cd') then
 . . .
else
 . . .
end if
```

B.5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \leq k \leq n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the call is executed.

B.6. Input/Output

B.6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)' ) x
```

B.6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=**, **err=**, and **iostat=** clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

B.6.3. Formatted I/O

B.6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2,'' isn''''t '',i1)' ) 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn''t ',i1)
```

and the character constant

```
 isn't
```

is copied into the output.

B.6.3.2. Positional Editing Codes

t, **tl**, **tr**, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. See section 3.4 in the main text.

B.6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first **write** statement prints:

```
hello there 12
```

while the second only prints

```
hello
```

B.6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. Since we don't normally put out optional plus signs, **ss** and **s** codes have the same effect in this implementation.

B.6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks are ignored following a **bn** code in a format statement, and are treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. Blanks are ignored by default.

B.6.3.6. Unrepresentable Values

The ANSI standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks.

B.6.3.7. Iw.m

There is a new integer output code, **i*w*.*m***. It is the same as **i*w***, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i*w*.0** is special, in that if the value being printed is 0, the output field is entirely blank. **i*w*.1** is the same as **i*w***.

B.6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e.** The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. We do not print it. There is a **g*w*.*d*** format code which is the same as **e*w*.*d*** and **f*w*.*d*** on input, but which chooses **f** or **e** formats for output depending on the size of the number and of *d*.

B.6.3.9. 'A' Format Code

A codes are used for character values. **a*w*** use a field width of *w*, while a plain **a** uses the length of the character item.

B.6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

B.6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*,  4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

B.6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a**

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

B.6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The ANSI standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**) There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,'(a)') x
read(x,'(i3,i4)') n1,n2
```

which reads a card image into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case, a record is a single element of an array of character strings.

B.6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

B.6.8.1. OPEN

The **open** statement connects a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 19. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write**

err= is the same as in **read** or **write**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status=scratch**

status= a character expression which evaluates to one of **'old'**, **'new'**, **'scratch'**, or **'unknown'**. If this parameter is not given, **'unknown'** is assumed. The meaning of **'unknown'** is processor dependent; this system treats it as synonymous with **'old'** If **'scratch'** is given, a temporary file is created. Temporary files are destroyed at the end of execution. If **'new'** is given, the file must not exist. It will be created for both reading and writing. If **'old'** is given, it is an error for the file not to exist.

access= a character expression which evaluates to **'sequential'** or **'direct'**, depending on whether the file is to be opened for sequential or direct I/O.

form= a character expression which evaluates to **'formatted'** or **'unformatted'**.

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= a character expression which evaluates to **'null'** or **'zero'**. This parameter has meaning only for formatted I/O. The default value is **null**; **zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

B.6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **'keep'** or **'delete'** Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

B.6.8.3. INQUIRE

The **inquire** statement gives information about a unit (inquire by unit) or a file (inquire by file). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which is assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.

formatted= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

unformatted= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the ANSI standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to use the *access*(3f) function. The **inquire** statement does not give a way of determining permissions.

Appendix C

Bibliography

1. *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.* New York, American National Standards Institute, 1978.
2. *USA Standard FORTRAN, USAS X3.9-1966,* New York: United States of America Standards Institute, March 7, 1966. Clarified in *Comm. ACM* **12**, *289 (1969) and Comm. ACM* **14**, *628 (1971).*
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall (1978).
5. S. C. Johnson, 'A Portable Compiler: Theory and Practice', Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).
7. B. W. Kernighan, 'RATFOR — A Preprocessor for a Rational Fortran', *Bell Laboratories Computing Science Technical Report #55,* (January 1977).

The following books or documents describe aspects of FORTRAN-77. This list is not necessarily complete. No particular endorsement is implied.

1. Brainerd, Walter S., et al. *FORTRAN-77 Programming.* Harper Row, 1978.
2. Day, A. C. *Compatible Fortran.* Cambridge University Press, 1979.
3. Dock, V. Thomas. *Structured FORTRAN-77 IV Programming.* West, 1979.
4. Feldman, S. I. 'The Programming Language EFL,' *Bell Laboratories Technical Report.* June 1979.
5. Hume, J. N., and R. C. Holt. *Programming FORTRAN-77.* Reston, 1979.
6. Katzan, Harry, Jr. *FORTRAN-77.* Van Nostrand-Reinhold, 1978.
7. Meissner, Loren P., and Organick, Elliott I. *FORTRAN-77 Featuring Structured Programming.* Addison-Wesley, 1979.
8. Merchant, Michael J. *ABC's of FORTRAN-77 Programming.* Wadsworth, 1979.
9. Page, Rex, and Richard Didday. *FORTRAN-77 for Humans.* West, 1980.
10. Wagener, Jerrold L. *Principles of FORTRAN-77 Programming.* Wiley, 1980.

A *FORTRAN Language Reference Manual*
is currently in preparation.

FORTRAN LIBRARY FUNCTIONS

The following collection of manual pages (section 3F) describes the functions from the Fortran run-time library.

NAME

intro – introduction to FORTRAN library functions

DESCRIPTION

This section describes those functions that are in the FORTRAN run time library. The functions listed here provide an interface from *f77* programs to the system in the same manner as the C library does for C programs. They are automatically loaded as needed by the Fortran compiler *f77*(1).

Most of these functions are in libU77.a. Some are in libF77.a or libI77.a. A few intrinsic functions are described for the sake of completeness.

For efficiency, the SCCS ID strings are not normally included in the *a.out* file. To include them, simply declare

```
external f77lid
```

in any *f77* module.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
abort	abort.3f	terminate abruptly with memory image
access	access.3f	determine accessability of a file
alarm	alarm.3f	execute a subroutine after a specified time
bessel functions		bessel.3fof two kinds for integer orders
bit	bit.3f	and, or, xor, not, rshift, lshift bitwise functions
chdir	chdir.3f	change default directory
chmod	chmod.3f	change mode of a file
ctime	time.3f	return system time
dflmax	range.3f	return extreme values
dflmin	range.3f	return extreme values
drand	rand.3f	return random values
dtime	etime.3f	return elapsed execution time
etime	etime.3f	return elapsed execution time
exit	exit.3f	terminate process with status
fdate	fdate.3f	return date and time in an ASCII string
fgetc	getc.3f	get a character from a logical unit
flmax	range.3f	return extreme values
flmin	range.3f	return extreme values
flush	flush.3f	flush output to a logical unit
fork	fork.3f	create a copy of this process
fpecnt	trpfpe.3f	trap and repair floating point faults
fputc	putc.3f	write a character to a FORTRAN logical unit
fseek	fseek.3f	reposition a file on a logical unit
fstat	stat.3f	get file status
ftell	fseek.3f	reposition a file on a logical unit
gerror	perror.3f	get system error messages
getarg	getarg.3f	return command line arguments
getc	getc.3f	get a character from a logical unit
getcwd	getcwd.3f	get pathname of current working directory
getenv	getenv.3f	get value of environment variables
getgid	getuid.3f	get user or group ID of the caller
getlog	getlog.3f	get user's login name
getpid	getpid.3f	get process id
getuid	getuid.3f	get user or group ID of the caller
gmtime	time.3f	return system time

hostnm	hostnm.3f	get name of current host
iargc	getarg.3f	return command line arguments
idate	idate.3f	return date or time in numerical form
ierrno	perror.3f	get system error messages
index	index.3f	tell about character objects
inmax	range.3f	return extreme values
ioinit	ioinit.3f	change f77 I/O initialization
irand	rand.3f	return random values
isatty	ttynam.3f	find name of a terminal port
itime	idate.3f	return date or time in numerical form
kill	kill.3f	send a signal to a process
len	index.3f	tell about character objects
link	link.3f	make a link to an existing file
lnblnk	index.3f	tell about character objects
loc	loc.3f	return the address of an object
long	long.3f	integer object conversion
lstat	stat.3f	get file status
ltime	time.3f	return system time
perror	perror.3f	get system error messages
putc	putc.3f	write a character to a FORTRAN logical unit
qsort	qsort.3f	quick sort
rand	rand.3f	return random values
rename	rename.3f	rename a file
rindex	index.3f	tell about character objects
short	long.3f	integer object conversion
signal	signal.3f	change the action for a signal
sleep	sleep.3f	suspend execution for an interval
stat	stat.3f	get file status
symlnk	link.3f	make a link to an existing file
system	system.3f	execute a UNIX command
tclose	topen.3f	f77 tape I/O
time	time.3f	return system time
topen	topen.3f	f77 tape I/O
tread	topen.3f	f77 tape I/O
trewin	topen.3f	f77 tape I/O
trpfpe	trpfpe.3f	trap and repair floating point faults
tskipf	topen.3f	f77 tape I/O
tstate	topen.3f	f77 tape I/O
ttynam	ttynam.3f	find name of a terminal port
twrite	topen.3f	f77 tape I/O
unlink	unlink.3f	remove a directory entry
wait	wait.3f	wait for a process to terminate

NAME

abort – terminate abruptly with memory image

SYNOPSIS

subroutine abort (string)
character*(*) string

DESCRIPTION

Abort cleans up the I/O buffers and then aborts producing a *core* file in the current directory. If *string* is given, it is written to logical unit 0 preceeded by "abort:".

FILES

/usr/lib/libF77.a

SEE ALSO

abort(3)

NAME

access – determine accessability of a file

SYNOPSIS

integer function access (name, mode)
character*(*) name, mode

DESCRIPTION

Access checks the given file, *name,* for accessability with respect to the caller according to *mode.* *Mode* may include in any order and in any combination one or more of:

r test for read permission

w test for write permission

x test for execute permission

(blank) test for existence

An error code is returned if either argument is illegal, or if the file can not be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

FILES

/usr/lib/libU77.a

SEE ALSO

access(2), perror(3F)

NAME

alarm – execute a subroutine after a specified time

SYNOPSIS

```
integer function alarm (time, proc)
integer time
external proc
```

DESCRIPTION

This routine arranges for subroutine *proc* to be called after *time* seconds. If *time* is "0", the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libU77.a

SEE ALSO

alarm(3C), sleep(3F), signal(3F)

BUGS

A subroutine cannot pass its own name to *alarm* because of restrictions in the standard.

NAME

chdir – change default directory

SYNOPSIS

integer function chdir (dirname)
character*(*) dirname

DESCRIPTION

The default directory for creating and locating files will be changed to *dirname*. Zero is returned if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(2), cd(1), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Use of this function may cause **inquire** by unit to fail.

Certain FORTRAN file operations reopen files by name. Using *chdir* while doing I/O may result in the run-time system to lose track of files created with relative pathnames (including files created by OPEN statements without file names).

NAME

chmod – change mode of a file

SYNOPSIS

integer function chmod (name, mode)
character*(*) name, mode

DESCRIPTION

This function changes the filesystem *mode* of file *name*. *Mode* can be any specification recognized by *chmod*(1). *Name* must be a single pathname.

The normal returned value is 0. Any other value will be a system error number.

FILES

/usr/lib/libU77.a
/bin/chmod exec'ed to change the mode.

SEE ALSO

chmod(1)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

exit – terminate process with status

SYNOPSIS

subroutine exit (status)
integer status

DESCRIPTION

Exit flushes and closes all the process's files, and notifies the parent process if it is executing a *wait*. The low-order 8 bits of *status* are available to the parent process. (Therefore *status* should be in the range 0 – 255)

This call will never return.

The C function *exit* may cause cleanup actions before the final 'sys exit'.

FILES

/usr/lib/libF77.a

SEE ALSO

exit(2), fork(2), fork(3f), wait(2), wait(3f)

NAME

flush – flush output to a logical unit

SYNOPSIS

subroutine flush (lunit)

DESCRIPTION

Flush causes the contents of the buffer for logical unit *lunit* to be flushed to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

FILES

/usr/lib/libI77.a

SEE ALSO

fclose(3S)

NAME

fork – create a copy of this process

SYNOPSIS

integer function fork()

DESCRIPTION

Fork creates a copy of the calling process. The only distinction between the 2 processes is that the value returned to one of them (referred to as the 'parent' process) will be the process id if the copy. The copy is usually referred to as the 'child' process. The value returned to the 'child' process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and will be the negation of the system error code. See perror(3F).

A corresponding *exec* routine has not been provided because there is no satisfactory way to retain open logical units across the exec. However, the usual function of *fork/exec* can be performed using *system*(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

fork(2), wait(3F), kill(3F), system(3F), perror(3F)

NAME

fseek, ftell – reposition a file on a logical unit

SYNOPSIS

integer function fseek (lunit, offset, from)
integer offset, from

integer function ftell (lunit)

DESCRIPTION

lunit must refer to an open logical unit. *offset* is an offset in bytes relative to the position specified by *from*. Valid values for *from* are:

0 meaning 'beginning of the file'
1 meaning 'the current position'
2 meaning 'the end of the file'

The value returned by *fseek* will be 0 if successful, a system error code otherwise. (See perror(3F))

Ftell returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See perror(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

fseek(3S), perror(3F)

NAME

getarg, iargc – return command line arguments

SYNOPSIS

```
subroutine getarg (k, arg)
character*(*) arg

function iargc ()
```

DESCRIPTION

A call to *getarg* will return the *kth* command line argument in character string *arg*. The *0th* argument is the command name.

Iargc returns the index of the last command line argument.

FILES

/usr/lib/libU77.a

SEE ALSO

execve(2), getenv(3F)

NAME

getc, fgetc – get a character from a logical unit

SYNOPSIS

```
integer function getc (char)
character char

integer function fgetc (lunit, char)
character char
```

DESCRIPTION

These routines return the next character from a file associated with a fortran logical unit, bypassing normal fortran I/O. *Getc* reads from logical unit 5, normally connected to the control terminal input.

The value of each function is a system status code. Zero indicates no error occured on the read; -1 indicates end of file was detected. A positive value will be either a UNIX system error code or an f77 I/O error code. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

getc(3S), intro(2), perror(3F)

NAME

getcwd – get pathname of current working directory

SYNOPSIS

integer function getcwd (dirname)
character*(*) dirname

DESCRIPTION

The pathname of the default directory for creating and locating files will be returned in *dirname*. The value of the function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(3F), perror(3F), getwd(3)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

getenv – get value of environment variables

SYNOPSIS

subroutine getenv (ename, evalue)
character*(*) ename, evalue

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *ename*=*value* and returns *value* in *evalue* if such a string is present, otherwise fills *evalue* with blanks.

FILES

/usr/lib/libU77.a

SEE ALSO

execve(2), environ(5)

NAME

getfd – get the file descriptor of an external unit number

SYNOPSIS

```
integer function getfd(unitn)
integer unitn
```

DESCRIPTION

Getfd **returns the 'file descriptor' of an external unit number if the unit is connected and –1 otherwise.**

FILES

/usr/lib/libI77.a

SEE ALSO

open(2)

NAME

getlog – get user's login name

SYNOPSIS

```
subroutine getlog (name)
character*(*) name

character*(*) function getlog()
```

DESCRIPTION

Getlog will return the user's login name or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libU77.a

SEE ALSO

getlogin(3)

NAME

getpid – get process id

SYNOPSIS

integer function getpid()

DESCRIPTION

Getpid **returns the process ID number of the current process.**

FILES

/usr/lib/libU77.a

SEE ALSO

getpid(2)

NAME

getuid, getgid – get user or group ID of the caller

SYNOPSIS

```
integer function getuid()

integer function getgid()
```

DESCRIPTION

These functions return the real user or group ID of the user of the process.

FILES

/usr/lib/libU77.a

SEE ALSO

getuid(2)

NAME

hostnm – get name of current host

SYNOPSIS

integer function hostnm (name)
character*(*) name

DESCRIPTION

This function puts the name of the current host into character string *name*. The return value should be 0; any other value indicates an error.

FILES

/usr/lib/libU77.a

SEE ALSO

gethostname(2)

NAME

idate, itime – return date or time in numerical form

SYNOPSIS

```
subroutine idate (iarray)
integer iarray(3)

subroutine itime (iarray)
integer iarray(3)
```

DESCRIPTION

Idate returns the current date in *iarray*. The order is: day, mon, year. Month will be in the range 1-12. Year will be $\geq$ 1969.

Itime returns the current time in *iarray*. The order is: hour, minute, second.

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3F), fdate(3F)

NAME

index, rindex, lnblnk, len – tell about character objects

SYNOPSIS

(intrinsic) function index (string, substr)
character*(*) string, substr

integer function rindex (string, substr)
character*(*) string, substr

function lnblnk (string)
character*(*) string

(intrinsic) function len (string)
character*(*) string

DESCRIPTION

Index (rindex) returns the index of the first (last) occurrence of the substring *substr* in *string,* or zero if it does not occur. *Index* is an f77 intrinsic function; *rindex* is a library routine.

Lnblnk returns the index of the last non-blank character in *string.* This is useful since all f77 character objects are fixed length, blank padded. Intrinsic function *len* returns the size of the character object argument.

FILES

/usr/lib/libF77.a

NAME

ioinit – change f77 I/O initialization

SYNOPSIS

logical function loinit (cctl, bzro, apnd, prefix, vrbose)
logical cctl, bzro, apnd, vrbose
character*(*) prefix

DESCRIPTION

This routine will initialize several global parameters in the f77 I/O system, and attach externally defined files to logical units at run time. The effect of the flag arguments applies to logical units opened after *ioinit* is called. The exception is the preassigned units, 5 and 6, to which *cctl* and *bzro* will apply at any time. *Ioinit* is written in Fortran-77.

By default, carriage control is not recognized on any logical unit. If *cctl* is **.true.** then carriage control will be recognized on formatted output to all logical units except unit 0, the diagnostic channel. Otherwise the default will be restored.

By default, trailing and embedded blanks in input data fields are ignored. If *bzro* is **.true.** then such blanks will be treated as zero's. Otherwise the default will be restored.

By default, all files opened for sequential access are positioned at their beginning. It is sometimes necessary or convenient to open at the END-OF-FILE so that a write will append to the existing data. If *apnd* is **.true.** then files opened subsequently on any logical unit will be positioned at their end upon opening. A value of **.false.** will restore the default behavior.

Many systems provide an automatic association of global names with fortran logical units when a program is run. There is no such automatic association in f77. However, if the argument *prefix* is a non-blank string, then names of the form **prefix**NN will be sought in the program environment. The value associated with each such name found will be used to open logical unit NN for formatted sequential access. For example, if f77 program *myprogram* included the call

```
call ioinit (.true., .false., .false., 'FORT', .false.)
```

then when the following sequence

```
% setenv FORT01 mydata
% setenv FORT12 myresults
% myprogram
```

would result in logical unit 1 opened to file *mydata* and logical unit 12 opened to file *myresults*. Both files would be positioned at their beginning. Any formatted output would have column 1 removed and interpreted as carriage control. Embedded and trailing blanks would be ignored on input.

If the argument *vrbose* is **.true.** then *ioinit* will report on its activity.

The effect of

```
call ioinit (.true., .true., .false., '', .false.)
```

can be achieved without the actual call by including "–ll66" on the *f77* command line. This gives carriage control on all logical units except 0, causes files to be opened at their beginning, and causes blanks to be interpreted as zero's.

The internal flags are stored in a labeled common block with the following definition:

```
integer*2 ieof, ictl, ibzr
common /ioiflg/ ieof, ictl, ibzr
```

FILES

/usr/lib/libI77.a	f77 I/O library
/usr/lib/libI66.a	sets older fortran I/O modes

SEE ALSO

getarg(3F), getenv(3F), "Introduction to the f77 I/O Library"

BUGS

Prefix can be no longer than 30 characters. A pathname associated with an environment name can be no longer than 255 characters.

The "+" carriage control does not work.

NAME

kill – send a signal to a process

SYNOPSIS

function kill (pid, signum)
integer pid, signum

DESCRIPTION

Pid must be the process id of one of the user's processes. *Signum* must be a valid signal number (see signal(3)). The returned value will be 0 if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

kill(2), signal(3), signal(3F), fork(3F), perror(3F)

NAME

link, symlnk – make a link to an existing file

SYNOPSIS

function link (name1, name2)
character*(*) name1, name2

integer function symlnk (name1, name2)
character*(*) name1, name2

DESCRIPTION

Name1 must be the pathname of an existing file. *Name2* is a pathname to be linked to file *name1*. *Name2* must not already exist. The returned value will be 0 if successful; a system error code otherwise.

Symlnk creates a symbolic link to *name1*.

FILES

/usr/lib/libU77.a

SEE ALSO

link(2), symlink(2), perror(3F), unlink(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

loc – return the address of an object

SYNOPSIS

function loc (arg)

DESCRIPTION

The returned value will be the address of *arg*.

FILES

/usr/lib/libU77.a

NAME

perror, gerror, ierrno – get system error messages

SYNOPSIS

subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror()

function ierrno()

DESCRIPTION

Perror will write a message to fortran logical unit 0 appropriate to the last detected system error. *String* will be written preceding the standard error message.

Gerror returns the system error message in character variable *string*. *Gerror* may be called either as a subroutine or as a function.

Ierrno will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libU77.a

SEE ALSO

intro(2), perror(3), "Introduction to the f77 I/O Library"

BUGS

String in the call to *perror* can be no longer than 127 characters.

The length of the string returned by *gerror* is determined by the calling program.

NOTES

UNIX system error codes are described in *intro*(2). The f77 I/O error codes and their meanings are:

100	"error in format"
101	"illegal unit number"
102	"formatted io not allowed"
103	"unformatted io not allowed"
104	"direct io not allowed"
105	"sequential io not allowed"
106	"can't backspace file"
107	"off beginning of record"
108	"can't stat file"
109	"no * after repeat count"
110	"off end of record"
111	"truncation failed"
112	"incomprehensible list input"
113	"out of free space"
114	"unit not connected"
115	"read unexpected character"
116	"blank logical input field"
117	"'new' file exists"

118	"can't find 'old' file"
119	"unknown system error"
120	"requires seek ability"
121	"illegal argument"
122	"negative repeat count"
123	"illegal operation for unit"

NAME

putc, fputc – write a character to a FORTRAN logical unit

SYNOPSIS

integer function putc (char)
character char

integer function fputc (lunit, char)
character char

DESCRIPTION

These funtions write a character to the file associated with a FORTRAN logical unit bypassing normal FORTRAN I/O. *Putc* writes to logical unit 6, normally connected to the control terminal output.

The value of each function will be zero unless some error occurred; a system error code otherwise. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

putc(3S), intro(2), perror(3F)

NAME

qsort – quick sort

SYNOPSIS

subroutine qsort (array, len, isize, compar)
external compar
integer*2 compar

DESCRIPTION

One dimensional *array* contains the elements to be sorted. *len* is the number of elements in the array. *isize* is the size of an element, typically -

4 for **integer** and **real**
8 for **double precision** or **complex**
16 for **double complex**
(length of character object) for **character** arrays

Compar is the name of a user supplied integer*2 function that will determine the sorting order. This function will be called with 2 arguments that will be elements of *array*. The function must return -

negative if arg 1 is considered to precede arg 2
zero if arg 1 is equivalent to arg 2
positive if arg 1 is considered to follow arg 2

On return, the elements of *array* will be sorted.

FILES

/usr/lib/libU77.a

SEE ALSO

qsort(3)

NAME

flmin, flmax, dflmin, dflmax, inmax – return extreme values

SYNOPSIS

function flmin()

function flmax()

double precision function dflmin()

double precision function dflmax()

function inmax()

DESCRIPTION

Functions *flmin* and *flmax* return the minimum and maximum positive floating point values respectively. Functions *dflmin* and *dflmax* return the minimum and maximum positive double precision floating point values. Function *inmax* returns the maximum positive integer value.

These functions can be used by programs that must scale algorithms to the numerical range of the processor.

The values returned by *flmin* and *dflmin* are the smallest normalized IEEE format floating point values. The values returned by *flmax* and *dflmax* are the largest finite IEEE format floating point values.

The approximate values of these functions for the Sun Workstation are:

flmin 1.175494e–38

flmax 3.402823e+ 38

dflmin 2.2250738590e–308

dflmax
1.7976931349e+ 308

inmax 2147483647

FILES

/usr/lib/libF77.a

NAME

rename – rename a file

SYNOPSIS

integer function rename (from, to)
character*(*) from, to

DESCRIPTION

From must be the pathname of an existing file. *To* will become the new pathname for the file. If *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same filesystem. If *to* exists, it will be removed first.

The returned value will be 0 if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

rename(2), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

signal – change the action for a signal

SYNOPSIS

integer function signal(signum, proc, flag)
integer signum, flag
external proc

DESCRIPTION

When a process incurs a signal (see *signal*(3)) the default action is usually to clean up and abort. The user may choose to write an alternative signal handling routine. A call to *signal* is the way this alternate action is specified to the system.

Signum is the signal number (see *signal*(3)). If *flag* is negative, then *proc* must be the name of the user signal handling routine. If *flag* is zero or positive, then *proc* is ignored and the value of *flag* is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. Two possible values for *flag* have specific meanings: 0 means "use the default action" (See NOTES below), 1 means "ignore this signal".

A positive returned value is the previous action definition. A value greater than 1 is the address of a routine that was to have been called on occurrence of the given signal. The returned value can be used in subsequent calls to *signal* in order to restore a previous action definition. A negative returned value is the negation of a system error code. (See *perror*(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

kill(1), signal(3), kill(3F)

NOTES

f77 arranges to trap certain signals when a process is started. The only way to restore the default **f77** action is to save the returned value from the first call to *signal*.

If the user signal handler is called, it will be passed the signal number as an integer argument.

NAME

sleep – suspend execution for an interval

SYNOPSIS

subroutine sleep (itime)

DESCRIPTION

Sleep causes the calling process to be suspended for *itime* seconds. The actual time can be up to 1 second less than *itime* due to granularity in system timekeeping.

FILES

/usr/lib/libU77.a

SEE ALSO

sleep(3)

NAME

system – execute a UNIX command

SYNOPSIS

integer function system (string)
character*(*) string

DESCRIPTION

System causes *string* to be given to your shell as input as if the string had been typed as a command. If environment variable **SHELL** is found, its value will be used as the command interpreter (shell); otherwise *sh*(1) is used.

The current process waits until the command terminates. The returned value will be the exit status of the shell. See *wait*(2) for an explanation of this value.

FILES

/usr/lib/libU77.a

SEE ALSO

execve(2), wait(2), system(3)

BUGS

String can not be longer than NCARGS–50 characters, as defined in <sys/param.h>.

NAME

topen, tclose, tread, twrite, trewin, tskipf, tstate – f77 tape I/O

SYNOPSIS

```
integer function topen (tlu, devnam, label)
integer tlu
character*(*) devnam
logical label

integer function tclose (tlu)
integer tlu

integer function tread (tlu, buffer)
integer tlu
character*(*) buffer

integer function twrite (tlu, buffer)
integer tlu
character*(*) buffer

integer function trewin (tlu)
integer tlu

integer function tskipf (tlu, nfiles, nrecs)
integer tlu, nfiles, nrecs

integer function tstate (tlu, fileno, recno, errf, eoff, eotf, tcsr)
integer tlu, fileno, recno, tcsr
logical errf, eoff, eotf
```

DESCRIPTION

These functions provide a simple interface between f77 and magnetic tape devices. A "tape logical unit", *tlu*, is "topen"ed in much the same way as a normal f77 logical unit is "open"ed. All other operations are performed via the *tlu*. The *tlu* has no relationship at all to any normal f77 logical unit.

Topen associates a device name with a *tlu*. *Tlu* must be in the range 0 to 3. The logical argument *label* should indicate whether the tape includes a tape label. This is used by *trewin* below. *Topen* does not move the tape. The normal returned value is 0. If the value of the function is negative, an error has occured. See *perror*(3f) for details.

Tclose closes the tape device channel and removes its association with *tlu*. The normal returned value is 0. A negative value indicates an error.

Tread reads the next physical record from tape to *buffer*. *Buffer* must be of type **character**. The size of *buffer* should be large enough to hold the largest physical record to be read. The actual number of bytes read will be returned as the value of the function. If the value is 0, the end-of-file has been detected. A negative value indicates an error.

Twrite writes a physical record to tape from *buffer*. The physical record length will be the size of *buffer*. *Buffer* must be of type **character**. The number of bytes written will be returned. A value of 0 or negative indicates an error.

Trewin rewinds the tape associated with *tlu* to the beginning of the first data file. If the tape is a labelled tape (see *topen* above) then the label is skipped over after rewinding. The normal returned value is 0. A negative value indicates an error.

Tskipf allows the user to skip over files and/or records. First, *nfiles* end-of-file marks are skipped. If the current file is at EOF, this counts as 1 file to skip. (Note: This is the way to reset the EOF status for a *tlu*.) Next, *nrecs* physical records are skipped over. The normal returned value is 0. A negative value indicates an error.

Finally, *tstate* allows the user to determine the logical state of the tape I/O channel and to see the tape drive control status register. The values of *fileno* and *recno* will be returned and indicate the current file and record number. The logical values *errf*, *eoff*, and *eotf* indicate an error has occurred, the current file is at EOF, or the tape has reached logical end-of-tape. End-of-tape (EOT) is indicated by an empty file, often referred to as a double EOF mark. It is not allowed to read past EOT although it is allowed to write. The value of *tcsr* will reflect the tape drive control status register. See *tm*(4S) for details.

FILES

/usr/lib/libU77.a

SEE ALSO

tm(4S), perror(3f)

NAME

trpfpe, fpecnt – trap and repair floating point faults

SYNOPSIS

subroutine trpfpe (numesg, rtnval)
double precision rtnval

integer function fpecnt ()

common /fpeflt/ fperr
logical fperr

DESCRIPTION

NOTE: This routine applies only to Vax computers. It is a null routine on the PDP11.

Trpfpe sets up a signal handler to trap arithmetic exceptions. If the exception is due to a floating point arithmetic fault, the result of the operation is replaced with the *rtnval* specified. *Rtnval* must be a double precision value. For example, "0d0" or "dflmax()".

The first *numesg* occurrences of a floating point arithmetic error will cause a message to be written to the standard error file. Any exception that can't be repaired will result in the default action, typically an abort with core image.

Fpecnt returns the number of faults since the last call to *trpfpe*.

The logical value in the common block labelled *fpeflt* will be set to **.true.** each time a fault occurs.

FILES

/usr/lib/libF77.a

SEE ALSO

signal(3f), range(3f)

BUGS

This routine works only for *faults*, not *traps*. This is primarily due to the Vax architecture.

If the operation involves changing the stack pointer, it can't be repaired. This seldom should be a problem with the f77 compiler, but such an operation might be produced by the optomizer.

The POLY and EMOD opcodes are not dealt with.

NAME

ttynam, isatty – find name of a terminal port

SYNOPSIS

character*(*) function ttynam (lunit)

logical function isatty (lunit)

DESCRIPTION

Ttynam **returns a blank padded path name of the terminal device associated with logical unit** ***lunit*****.**

Isatty **returns .true. if** ***lunit*** **is associated with a terminal device, .false. otherwise.**

FILES

/dev/*
/usr/lib/libU77.a

DIAGNOSTICS

Ttynam **returns an empty string (all blanks) if** ***lunit*** **is not associated with a terminal device in directory '/dev'.**

NAME

unlink – remove a directory entry

SYNOPSIS

integer function unlink (name)
character*(*) name

DESCRIPTION

Unlink causes the directory entry specified by pathname *name* to be removed. If this was the last link to the file, the contents of the file are lost. The returned value will be zero if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

unlink(2), link(3F), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

NAME

wait – wait for a process to terminate

SYNOPSIS

integer function wait (status)
integer status

DESCRIPTION

Wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last *wait*, return is immediate; if there are no children, return is immediate with an error code.

If the returned value is positive, it is the process ID of the child and *status* is its termination status (see *wait*(2)). If the returned value is negative, it is the negation of a system error code.

FILES

/usr/lib/libU77.a

SEE ALSO

wait(2), signal(3F), kill(3F), perror(3F)

RATFOR

A Preprocessor for a Rational Fortran

Table of Contents

Ratfor — A Preprocessor for a Rational FORTRAN

Although FORTRAN is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of FORTRAN while retaining its desirable qualities, by providing decent control flow statements:

statement grouping
: using { and } in the style of C.

decision-making
: via **if-else** and **switch** statements.

looping constructs
: via **while**, **for**, **do**, and **repeat-until** statements.

controlled exits from loops
: via **break** and **next** statements.

and some 'syntactic sugar':

free form input
: multiple statements/line, automatic continuation

unobtrusive comments
: signalled by a # sign anywhere on the line.

translation
: of >, >=, etc., into .GT., .GE., etc.

return (expression)
: **statement for functions**

symbolic parameters
: via the **define** statement.

source file inclusion
: via the **include** statement.

Ratfor is implemented as a preprocessor which translates this language into FORTRAN.

Once the control flow and cosmetic deficiencies of FORTRAN are hidden, the resulting language is remarkably pleasant to use. *Ratfor* programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their FORTRAN equivalents.

It is readily possible to write *Ratfor* programs which are portable to other environments. *Ratfor* is written in itself in this way, so it is also portable; versions of *Ratfor* are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a FORTRAN preprocessor, the Ratfor language and its implementation, and user experience.

This paper is a revised and expanded version of one published in *Software — Practice and Experi-*

1. Introduction To Ratfor

Most programmers will agree that FORTRAN is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, FORTRAN is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable FORTRAN programs[1]. Finally, FORTRAN is often the most 'efficient' language available, particularly for programs requiring much computation.

But FORTRAN *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in FORTRAN are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one FORTRAN statement (with some *further* restrictions!). And of course there can be no ELSE part to a FORTRAN IF: there is no way to specify an alternative action if the IF is not satisfied.

The FORTRAN DO restricts the user to going forward in an arithmetic progression. It is fine for '1 to N in steps of 1 (or 2 or ...)', but there is no direct way to go backwards, or even (in ANSI FORTRAN[2]) to go from 1 to N-1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that FORTRAN programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with *Ratfor*. (The preprocessor idea is of course not new, and preprocessors for FORTRAN are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

1.1. Using the *Ratfor* Translator

Ratfor is the basic translator; it takes either a list of file names or the standard input and writes FORTRAN on the standard output. Options include **-6x**, which uses **x** as a continuation character in column 6 (UNIX uses **&** in column 1), and **-C**, which copies *Ratfor* comments into the generated FORTRAN.

Rc provides an interface to the *Ratfor* command which is much the same as *cc*. Thus

tutorial% **rc** [*options*] file ...

compiles the files specified by *files*. Files with names ending in *.r* are *Ratfor* source; other files are assumed to be for the loader. The flags **-C** and **-6x** described above are recognized, as are:

-c compile only; don't load

-f save intermediate FORTRAN *.f* files

-r Ratfor only; implies -c and -f

ence, October 1975.

–U flag undeclared variables (not universally available) Other flags are passed on to the loader.

2. Language Description

2.1. Design

Ratfor attempts to retain the merits of FORTRAN (universality, portability, efficiency) while hiding the worst FORTRAN inadequacies. The language *is* FORTRAN except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of *Ratfor* is to conceal this part of FORTRAN from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the 'cosmetic' deficiencies of FORTRAN, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — *Ratfor* does nothing about the host of other weaknesses of FORTRAN. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in *Ratfor* and what should not has been *Ratfor doesn't know any FORTRAN.* Any language feature which would require that *Ratfor* really understand FORTRAN has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the *Ratfor* language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

2.2. Statement Grouping

FORTRAN provides no way to group statements together, short of making them into a subroutine. The standard construction 'if a condition is true, do this group of things,' for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in FORTRAN. Instead a programmer is forced to translate this relatively clear thought into murky FORTRAN, by stating the negative condition and branching around the group of statements:

```
      if (x .le. 100) goto 10
            call error(5hx>100)
            err = 1
            return
10    ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in *Ratfor*. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single *Ratfor* statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have FORTRAN meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character '>' is clearer than '.GT.', so *Ratfor* translates it appropriately, along with several other similar shorthands. Although many FORTRAN compilers permit character strings in quotes (like """x>100"""), quotes are not allowed in ANSI FORTRAN, so *Ratfor* converts it into the right number of H 's: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
      call error("x>100")
      err = 1
      return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
      write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general *Ratfor* continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

2.3. The 'else' Clause

Ratfor provides an "**else**" statement to handle the construction 'if a condition is true, do this thing, *otherwise* do that thing.'

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The FORTRAN equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the FORTRAN version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the *Ratfor* version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal FORTRAN condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal FORTRAN condition* is anything that can legally go into a FORTRAN Logical IF. Ratfor does not check this clause, since it does not know enough FORTRAN to know what is permitted. The *Ratfor statement* is any *Ratfor* or FORTRAN statement, or any collection of them in braces.

2.4. Nested if's

Since the statement that follows an **if** or an **else** can be any *Ratfor* statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to −1 if **x** is less than zero, to +1 if **x** is greater than 100, and to 0 otherwise. Then in *Ratfor*, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight FORTRAN will necessarily be indirect because FORTRAN does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in *Ratfor*. In general the structure

```
if (...)
        ---
else if (...)
        ---
else if (...)
        ---
 ...
else
        ---
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the 'default' case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

2.5. if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else** 's. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in *Ratfor*, as it is in many other programming languages. The ambiguity is resolved in *Ratfor* (as elsewhere) by saying that in such cases the **else** goes with the closest previous **else 'ed un- if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must write*

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

2.6. The 'switch' Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {

        case expr1 :
                statements
        case expr2, expr3 :
                statements
        ...
        default:
                statements
}
```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression*, and there is a **default** section, the statements with it are done; if there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch** .)

2.7. The 'do' Statement

The **do** statement in *Ratfor* is quite similar to the DO statement in FORTRAN, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10      continue
```

The syntax is:

do *legal-FORTRAN-DO-text*
    Ratfor statement

The part that follows the keyword **do** has to be something that can legally go into a FORTRAN DO statement. Thus if a local version of FORTRAN allows DO limits to be expressions (which is not currently permitted in ANSI FORTRAN), they can be used in a Ratfor **do.**

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array m to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of m to –1, the diagonal to zero, and the lower triangle to +1. (The operator == is 'equals', that is, '.EQ.'.) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

2.8. 'break' and 'next'

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. "**break**" causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
    if (x(i) < 0.0)
        next
    process positive element
}
```

break and **next** also work in the other *Ratfor* looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

2.9. The 'while' Statement

One of the problems with the FORTRAN DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with I set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
    do i = j, k {
        ---
    }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the FORTRAN DO, it is that much harder to write and understand.

To overcome these difficulties, *Ratfor* provides a **while** statement, which is simply a loop: 'while some condition is true, repeat this group of statements'. It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
    # returns sin(x) to accuracy e, by
    # sin(x) = x - x**3/3! + x**5/5! - ...

    sin = x
    term = x

    i = 3
    while (abs(term)>e & i<100) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
        i = i + 2
    }

    return
    end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character '**#**' in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with FORTRAN's 'C in column 1' convention. Blank lines are also permitted anywhere (they are not in FORTRAN); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```
while (legal FORTRAN condition)
    Ratfor statement
```

As with the **if**, *legal FORTRAN condition* is something that can go into a FORTRAN Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by FORTRAN programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
    ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in FORTRAN as

```
100  if (nextch(ich) .eq. iblank) goto 100
```

but many FORTRAN programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

2.10. The 'for' Statement

The **for** statement is another *Ratfor* loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do.**

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+ 2) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
}
```

The syntax of the **for** statement is

```
for ( init ; condition ; increment )
        Ratfor statement
```

init is any single FORTRAN statement, which gets done once before the loop begins. *increment* is any single FORTRAN statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init*, *condition*, and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so "**for(;;)**" is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) != blank)
                break
```

('!=' is the same as '.NE.'). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. **break** and and **next** work in **for**'s and **while**'s just as in **do**'s. If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular FORTRAN DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. Forgetting this is a common error. Thus:

```
    DO 10 J = 1, 80
        I = 81 - J
        IF (CARD(I) .NE. BLANK) GO TO 11
10  CONTINUE
    I = 0
11  ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

2.11. The 'repeat-until' statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

repeat
    Ratfor statement
until (*legal FORTRAN condition*)

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

2.12. More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

2.13. 'return' Statement

The standard FORTRAN mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical,

and zero if they differ. The array ends are marked by the special value -1.

```
# equal — compare str1 to str2;
#   return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1) {
            equal = 1
            return
        }
    equal = 0
    return
    end
```

In many languages (e.g., PL/I) one instead says

return (*expression*)

to return a value from a function. Since this is often clearer, *Ratfor* provides such a **return** statement — in a function F, **return (expression)** is equivalent to

{ F = expression; return }

For example, here is **equal** again:

```
# equal — compare str1 to str2;
#   return 1 if equal, 0 if not
    integer function equal(str1, str2)
    integer str1(100), str2(100)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == -1)
            return(1)
    return(0)
    end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

2.14. Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

2.15. Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if *Ratfor* can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + - * , | & (_

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a FORTRAN label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

2.16. Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing FORTRAN program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

center box tab (/) ; c c c c c l c l .

character/translation/character/translation

==/.eq./!=/.ne. >/.gt./>=/.ge. </.lt./<=/.le. &/.and./|/.or. !/.not./^/.not.

In addition, the following translations are provided for input devices with restricted character sets.

center box tab (/) ; c c c c c l c l .

character/translation/character/translation

[/{/]/} $(/{/$)/}

2.17. 'define' Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define      ROWS    100
define      COLS    50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define  YES     1
define  NO      0
define  EOS     -1
define  ARB     100

# equal — compare str1 to str2;
#   return YES if equal, NO if not
    integer function equal(str1, str2)
    integer str1(ARB), str2(ARB)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == EOS)
            return(YES)
    return(NO)
    end
```

2.18. 'include' Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the *Ratfor* input in place of the include

statement. The standard usage is to place COMMON blocks on a file, and include that file whenever a copy is needed:

```
subroutine x
      include commonblocks
      ...
      end

suroutine y
      include commonblocks
      ...
      end
```

This ensures that all copies of the COMMON blocks are identical

2.19. Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, else clauses without an if, and most errors involving missing parentheses in statements. Beyond that, since *Ratfor* knows no FORTRAN, any errors you make will be reported by the FORTRAN compiler, so you will from time to time have to relate a FORTRAN diagnostic back to the *Ratfor* source.

Keywords are reserved — using if, else, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The FORTRAN nH convention is not recognized anywhere by *Ratfor*; use quotes instead.

3. Implementation

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The *Ratfor* grammar is simple and straightforward, being essentially

```
prog : stat
     | prog  stat
stat : if (...) stat
     | if (...) stat else stat
     | while (...) stat
     | for (...; ...; ...) stat
     | do ... stat
     | repeat stat
     | repeat stat until (...)
     | switch (...) { case ...: prog ...
                      default: prog }
     | return
     | break
     | next
     | digits  stat
     | { prog }
     | anything unrecognizable
```

The observation that *Ratfor* knows no FORTRAN follows directly from the rule that says a

statement is 'anything unrecognizable'. In fact most of FORTRAN falls into this category, since any statement that does not begin with one of the keywords is by definition 'unrecognizable.'

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L     continue
```

is generated, unless there is an **else** clause, in which case the code is

```
      goto L+1
L     continue
```

In this latter case, the code

```
L+1 continue
```

is produced after the *statement* part of the **else**. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
      if (.not. (i .gt. 0)) goto 100
      x = a
100   continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of 'inefficiency' will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of *Ratfor* is used on UNIX. C compilers are not as widely available as FORTRAN, however, so there is also a *Ratfor* written in itself and originally bootstrapped with the C version. The *Ratfor* version was written so as to translate into the portable subset of FORTRAN described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v\pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. *Ratfor* itself will not gratuitously generate non-standard FORTRAN.

The *Ratfor* version is about 1500 lines of *Ratfor* (compared to about 1000 lines of C); this compiles into 2500 lines of FORTRAN. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the *Ratfor* version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. Experience

4.1. Good Things

'It's so much better than FORTRAN' is the most common response of users when asked how well *Ratfor* meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts FORTRAN from a bad language into quite a reasonable one, assuming that FORTRAN data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in FORTRAN. More important, debugging and subsequent revision are much faster than in FORTRAN. Partly this is simply because the code can be read. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in *Ratfor* tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of FORTRAN's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a *Ratfor* implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of *Ratfor*, including a subset of the *Ratfor* preprocessor itself, can be found in [8].

4.2. Bad Things

The biggest single problem is that many FORTRAN syntax errors are not detected by *Ratfor* but by the local FORTRAN compiler. The compiler then prints a message in terms of the generated FORTRAN, and in a few cases this may be difficult to relate back to the offending *Ratfor* line, especially if the implementation conceals the generated FORTRAN. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but

this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since *Ratfor* generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the FORTRAN. Furthermore, there has been a steady improvement in *Ratfor*'s ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard FORTRAN constructions are not accepted by *Ratfor*, and this is perceived as a problem by users with a large corpus of existing FORTRAN programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary FORTRAN programs into *Ratfor*.

Users who export programs often complain that the generated FORTRAN is 'unreadable' because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy *Ratfor* comments into the generated FORTRAN), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since *Ratfor* is relatively easy to modify, there are now several dialects of *Ratfor*. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. Conclusions

Ratfor demonstrates that with modest effort it is possible to convert FORTRAN from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in 'features' — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for *Ratfor* to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Appendix A. Acknowledgements

C. A. R. Hoare once said that 'One thing [the language designer] should not do is to include untried ideas of his own.' *Ratfor* follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of *Ratfor* led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into FORTRAN for the first *Ratfor* version of *Ratfor*.

Appendix B. Bibliography

[1] B. G. Ryder, 'The PFORT Verifier,' *Software—Practice & Experience*, October 1974.

[2] American National Standard FORTRAN. American National Standards Institute, New York, 1966.

[3] *For-word: FORTRAN Development Newsletter*, August 1975.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[5] D. M. Ritchie and K. L. Thompson, 'The UNIX Time-sharing System.' *CACM*, July 1974.

[6] S. C. Johnson, 'YACC — Yet Another Compiler-Compiler.' Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, 'Structured Programming with goto Statements.' *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[9] B. S. Baker, 'Struct — A Program which Structures FORTRAN', Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, 'The Altran System for Rational Function Manipulation — A Survey.' *CACM*, August 1971.

Revision: C of 7th January 1984
For: Sun System Release 1.1

Berkeley Pascal User's Manual

for the

Sun Workstation

Sun Microsystems, Inc.,
2550 Garcia Avenue,
Mountain View,
California 94043
(415) 960-1300

Acknowledgements

Material in this *Berkeley Pascal User's Manual* was originally produced by William N. Joy, Susan L. Graham, Charles B. Haley Marshall Kirk McKusick, and Peter B. Kessler of the Computer Science Division, Department of Electrical Engineering and Computer Science, at the University of California at Berkeley.

The financial support of the first and second authors' work by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and the first author's work by an IBM Graduate Fellowship are gratefully acknowledged.

History of the implementation

The first Berkeley system was written by Ken Thompson in early 1976. The main features of the present system were implemented by Charles Haley and William Joy during the latter half of 1976. Earlier versions of this system have been in use since January, 1977.

The system was moved to the VAX-11 by Peter Kessler and Kirk McKusick with the porting of the interpreter in the spring of 1979, and the implementation of the compiler in the summer of 1980.

The whole system was moved to the Sun Workstation in 1983 by Peter Kessler and Kirk McKusick.

Copyrights

Revision History

Rev	Date	Comments
A	October 1980	This revision was the Berkeley Pascal User's Manual Version 2.0.
B	July 1983	This revision was the Berkeley Pascal User's Manual Version 3.0. Sun Microsystems updates to this manual removed references to the CYBER 6000 implementation details.
C	7 January 1984	Minor corrections and updates.

Preface

Berkeley Pascal is designed for interactive instructional use and runs on the Sun Workstation, the PDP/11 and VAX/11 computers. Interpretive code is produced, providing fast translation at the expense of slower execution speed. There is also a fully compatible compiler for the Sun Workstation and for the VAX/11. An execution profiler and Wirth's cross reference program are also available with the system.

The system supports full Pascal. The language accepted is 'standard' Pascal, and a small number of extensions. There is an option to suppress the extensions. The extensions include a separate compilation facility and the ability to link to object modules produced from other source languages.

The *User's Manual* gives basic usage examples for the Pascal components *pi*, *px*, *pix*, *pc*, and *pxp*. Errors commonly encountered in these programs are discussed. Details are given of special considerations due to the interactive implementation. A number of examples are provided including many dealing with input/output. An appendix supplements Wirth's *Pascal Report* to form the full definition of the Berkeley implementation of the language.

The Berkeley Pascal *User's Manual* consists of four major sections and an appendix. Section 1 introduces the Berkeley implementation and provides a number of tutorial examples. Section 2 discusses the error diagnostics produced by the translators *pc* and *pi*, and the runtime interpreter *px*. Section 3 describes input/output with special attention given to features of the interactive implementation and to features unique to UNIX.† Section 4 gives details on the components of the system and explanation of all relevant options. The *User's Manual* concludes with an appendix to Wirth's *Pascal Report* with which it forms a precise definition of the implementation.

† UNIX is a trademark of Bell Laboratories.

Table of Contents

Chapter 1

Basic UNIX Pascal

The following sections explain the basics of using Berkeley Pascal. In examples here we use the *ex*(1) text editor. Users of the *ed* text editor should have little trouble following these examples, as *ex* is similar to *ed*. We use *ex* because we can make clearer examples. Users with Sun Workstations should find *vi* (visual editor) more pleasant to use; we do not show its use here because its display-oriented nature makes it difficult to illustrate. The new UNIX user will find it helpful to read one of the introductory chapters on text editors, either in the Sun *Beginner's Guide to the Sun Workstation*, or in the *Editing and Text Processing* manual before continuing with this section.

1.1. A First Program

To prepare a program for Berkeley Pascal we first need to have an account on UNIX and to 'login' to the system on this account. These procedures are described in the *Beginner's Guide.*

Once we are logged in, we need to choose a name for our program; let's call it *first* since this is the first example. We must also choose a name for the file in which the program will be stored. The Berkeley Pascal system requires that programs reside in files which have names ending with the sequence *.p* so we call our file *first.p*.

A sample editing session to create this file would begin:

```
tutorial% ex first.p
"first.p" [New file]
:
```

We didn't expect the file to exist, so the 'New file' message doesn't bother us. *ex* now knows the name of the file we are creating. The ':' prompt indicates that *ex* is ready for command input. We can add the text for our program using the 'append' command as follows.

```
:append
program first(output)
begin
    writeln('Hello, world!')
end.
.
:
```

The line containing the single '.' character here indicated the end of the appended text. The ':' prompt indicates that *ex* is ready for another command. As *ex* operates in a temporary work

space we must now store the contents of this work space in the file *first.p* so we can use the Pascal translator and executor *pix* on it.

```
: write
"first.p" [New file] 4 lines, 59 characters
: quit
tutorial%
```

We wrote out the file from the edit buffer here with the 'write' command, and *ex* indicated the number of lines and characters written. We then quit the editor, and now have a prompt from the shell.‡

We are ready to try to translate and execute our program.

```
tutorial% pix first.p
Wed Oct 29 17:11 1980 first.p:
   2 begin
e ----- ↑ --- Inserted ' ; '
Execution begins . . .
Hello, world!
Execution terminated.

1 statements executed in 0000.40 seconds cpu time.
tutorial%
```

The translator first printed a syntax error diagnostic. The number 2 here indicates that the rest of the line is an image of the second line of our program. The translator is saying that it expected to find a ';' before the keyword **begin** on this line. If we look at the Pascal syntax charts in the Jensen-Wirth *User Manual*, or at some of the sample programs therein, we see that we have omitted the terminating ';' of the **program** statement on the first line of our program.

One other thing to notice about the error diagnostic is the letter 'e' at the beginning. It stands for 'error', indicating that our input was not legal Pascal. The fact that it is an 'e' rather than an 'E' indicates that the translator managed to recover from this error well enough that generation of code and execution could take place. Execution is possible whenever no fatal 'E' errors occur during translation. The other classes of diagnostics are 'w' warnings, which do not necessarily indicate errors in the program, but point out inconsistencies which are likely to be due to program bugs, and 's' standard-Pascal violations.†

After completing the translation of the program to interpretive code, the Pascal system indicates that execution of the translated program began. The output from the execution of the program then appeared. At program termination, the Pascal runtime system indicated the number of statements executed, and the amount of cpu time used, with the resolution of the latter being 1/60'th of a second.

Let us now fix the error in the program and translate it to a permanent object code file *obj* using *pi*. The program *pi* translates Pascal programs but stores the object code instead of executing it‡.

---

‡ Our examples here assume you are using *csh*.

†The standard Pascal warnings occur only when the associated s translator option is enabled. The s option is discussed in the *Options* section, below. Warning diagnostics are discussed near the end of the *Error Diagnostics* section, below; the associated w option is described in the *Options common to Pi, Pc, and Px* section.

‡This script indicates some other useful approaches to debugging Pascal programs. As in *ed* we can

```
tutorial% ex first.p
"first.p" 4 lines, 59 characters
:1 print
program first(output)
:s/$/;
program first(output);
:write
"first.p" 4 lines, 60 characters
:quit
tutorial% pi first.p
tutorial%
```

If we now use the UNIX *ls* list files command we can see what files we have:

```
tutorial% ls
first.p
obj
tutorial%
```

The file 'obj' here contains the Pascal interpreter code. We can execute this by typing:

```
tutorial% px obj
Hello, world!

1 statements executed in 0.02 seconds cpu time.
```

Alternatively, the command:

```
tutorial% obj
```

has the same effect. Some examples of different ways to execute the program follow.

```
tutorial% px
Hello, world!

1 statements executed in 0.02 seconds cpu time.
tutorial% pi -p first.p
tutorial% px obj
Hello, world!
tutorial% px -p first.p
Hello, world!
tutorial%
```

Note that *px* assumes that 'obj' is the file we wish to execute if we don't tell it otherwise. The last two translations use the -p no-post-mortem option to eliminate execution statistics and 'Execution begins' and 'Execution terminated' messages. See the *Options common to Pi, Pc, and Px* section for more details. If we now look at the files in our directory we will see:

---

shorten commands in *ex* to an initial prefix of the command name as we did with the *substitute* command here. We have also used the '!' shell escape command here to execute other commands with a shell without leaving the editor.

```
tutorial% ls
first.p
obj
tutorial%
```

We can give our object program a name other than 'obj' by using the move command *mv*(1). Thus to name our program 'hello':

```
tutorial% mv obj hello
tutorial% hello
Hello, world!
tutorial% ls
first.p
hello
tutorial%
```

Finally we can get rid of the Pascal object code by using the *rm*(1) remove file command:

```
tutorial% rm hello
tutorial% ls
first.p
tutorial%
```

For small programs which are being developed *pix* tends to be more convenient to use than *pi* and *px*. Except for absence of the *obj* file after a *pix* run, a *pix* command is equivalent to a *pi* command followed by a *px* command. For larger programs, where a number of runs testing different parts of the program are to be made, *pi* is useful as this *obj* file can be executed any desired number of times.

1.2. A Larger Program

Suppose that we have used *ex* to put a larger program in the file 'bigger.p'. We can list this program with line numbers by using the *cat*(1) command with the – n option:

```
tutorial% cat -n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7          d = 0.0625;   (* 1/16, 16 lines for interval [x, x+ 1] *)
     8          s = 32;       (* 32 character width for interval [x, x+ 1]
     9          h = 34;       (* Character position of x-axis *)
    10          c = 6.28138;  (* 2 * pi *)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim begin
    17                  x := d / i;
```

```
    18                  y := exp(-x9 * sin(i * x);
    19                  n := Round(s * y) + h;
    20                  repeat
    21                          write(' ');
    22                          n := n - 1
    23                  writeln('*')
    24  end.
tutorial%
```

This program is similar to program 4.9 on page 30 of the Jensen-Wirth *User Manual*. A number of problems have been introduced into this example for pedagogical reasons.

If we attempt to translate and execute the program using *pix* we get the following response:

```
tutorial% pix bigger.p
Wed Nov  9 15:53 1983  bigger.p:
   9        h = 34;      (* Character position of x-axis *)
w ----------------------------↑--- (* in a (* ... *) comment
   16       for i := 0 to lim begin
e ------------------------------↑---- Inserted keyword do
   18               y := exp(-x9 * sin(i * x);
E ------------------------------↑---- Undefined variable
e ------------------------------------------------↑---- Inserted ')'
   19               n := Round(s * y) + h;
E -------------------------↑---- Undefined function
E ------------------------------------------------↑---- Undefined variable
   23               writeln('*')
e ------------------↑---- Inserted ';'
   24  end.
E -----↑---- Expected keyword until
E ----------↑---- Malformed declaration
   ----------↑---- Unexpected end-of-file - QUIT
Execution suppressed due to compilation errors
tutorial%
```

Since there were fatal 'E' errors in our program, no code was generated and execution was necessarily suppressed. One thing which would be useful at this point is a listing of the program with the error messages. We can get this by using the command:

```
tutorial% pi -l bigger.p
```

There is no point in using *pix* here, since we know there are fatal errors in the program. This command will produce the output at our terminal. If we are at a terminal which does not produce a hard copy we may wish to print this listing off-line on a line printer. We can do this with the command:

```
tutorial% pi -l bigger.p | lpr
```

In the next few sections we will illustrate various aspects of the Berkeley Pascal system by correcting this program.

1.3. Correcting the First Errors

Most of the errors which occurred in this program were *syntactic* errors, those in the format and structure of the program rather than its content. Syntax errors are flagged by printing the offending line, and then a line which flags the location at which an error was detected. The flag line also gives an explanation stating either a possible cause of the error, a simple action which can be taken to recover from the error so as to be able to continue the analysis, a symbol which was expected at the point of error, or an indication that the input was 'malformed'. In the last case, the recovery may skip ahead in the input to a point where analysis of the program can continue.

In this example, the first error diagnostic indicates that the translator detected a comment within a comment. While this is not considered an error in 'standard' Pascal, it usually corresponds to an error in the program which is being translated. In this case, we have accidentally omitted the trailing '*)' of the comment on line 8. We can begin an editor session to correct this problem by doing:

```
tutorial% ex bigger.p
"bigger.p" 24 lines, 512 characters
:8s/$/ *)
        s := 32;        (* 32 character width for interval [x, x+ 1] *)
:
```

The second diagnostic, given after line 16, indicates that the keyword **do** was expected before the keyword **begin** in the **for** statement. If we examine the *statement* syntax chart on page 118 of the Jensen-Wirth *User Manual* we will discover that **do** is a necessary part of the **for** statement. Similarly, we could have referred to section C.3 of the Jensen-Wirth *User Manual* to learn about the **for** statement and gotten the same information there. It is often useful to refer to these syntax charts and to the relevant sections of this book.

We can correct this problem by first scanning for the keyword **for** in the file and then substituting the keyword **do** to appear in front of the keyword **begin** there. Thus:

```
:/for
        for i := 0 to lim begin
:s/begin/do &
        for i := 0 to lim do begin
:
```

The next error in the program is easy to pinpoint. On line 18, we didn't hit the shift key and got a '9' instead of a ')'. The translator diagnosed that 'x9' was an undefined variable and, later, that a ')' was missing in the statement. It should be stressed that *pi* is not suggesting that you should insert a ')' before the ';'. It is only indicating that making this change will help it to be able to continue analyzing the program so as to be able to diagnose further errors. You must then determine the true cause of the error and make the appropriate correction to the source text.

This error also illustrates the fact that one error in the input may lead to multiple error diagnostics. *Pi* attempts to give only one diagnostic for each error, but single errors in the input sometimes appear to be more than one error. It is also the case that *pi* may not detect an error when it occurs, but may detect it later in the input. This would have happened in this example if we had typed 'x' instead of 'x9'.

The translator next detected, on line 19, that the function *Round* and the variable *h* were undefined. It does not know about *Round* because Berkeley Pascal normally distinguishes between upper and lower case.† On UNIX lower-case is preferred‡, and all keywords and built-in **procedure** and **function** names are composed of lower-case letters, just as they are in the Jensen-Wirth *Pascal Report*. Thus we need to use the function *round* here. As far as *h* is concerned, we can see why it is undefined if we look back to line 9 and note that its definition was lost in the non-terminated comment. This diagnostic need not, therefore, concern us.

The next error which occurred in the program caused the translator to insert a ';' before the statement calling *writeln* on line 23. If we examine the program around the point of error we will see that the actual error is that the keyword **until** and an associated expression have been omitted here. Note that the diagnostic from the translator does not indicate the actual error, and is somewhat misleading. The translator made the correction which seemed to be most plausible. As the omission of a ';' character is a common mistake, the translator chose to indicate this as a possible fix here. It later detected that the keyword **until** was missing, but not until it saw the keyword **end** on line 24. The combination of these diagnostics indicate to us the true problem.

The final syntactic error message indicates that the translator needed an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, we can infer that another **begin** must have been mismatched, and have matched this **end**. Thus we see that we need an **end** to match the **begin** at line 16, and to appear before the final **end**. We can make these corrections:

```
:/x9/s//x)
            y := exp(-x) * sin(i * x);
:+ s/Round/round
            n := round(s * y) + h;
:/write
                  write(' ');
:/
            writeln('*')
:insert
            until n = 0;
.
:$
end.
:insert
      end
.
:
```

At the end of each **procedure** or **function** and the end of the **program** the translator summarizes references to undefined variables and improper usages of variables. It also gives warnings about potential errors. In our program, the summary errors do not indicate any further problems but the warning that *c* is unused is somewhat suspicious. Examining the program we see that the constant was intended to be used in the expression which is an argument to *sin*, so we can correct this expression, and translate the program. We have now made a correction for each diagnosed error in our program.

†In "standard" Pascal no distinction is made based on case.
‡One good reason for using lower-case is that it is easier to type.

```
:?i ?s//c /
            y := exp(-x) * sin(c * x);
:write
"bigger.p" 26 lines, 538 characters
:quit
tutorial% pi bigger.p
tutorial%
```

It should be noted that the translator suppresses warning diagnostics for a particular **procedure**, **function** or the main **program** when it finds severe syntax errors in that part of the source text. This is to prevent possibly confusing and incorrect warning diagnostics from being produced. Thus these warning diagnostics may not appear in a program with bad syntax errors until these errors are corrected.

We are now ready to execute our program for the first time. We will do so in the next section after giving a listing of the corrected program for reference purposes.

```
tutorial% cat -n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7            d = 0.0625;   (* 1/16, 16 lines for interval [x, x+ 1] *)
     8            s = 32;       (* 32 character width for interval [x, x+ 1] *)
     9            h = 34;       (* Character position of x-axis *)
    10            c = 6.28138;  (* 2 * pi *)
    11            lim = 32;
    12  var
    13            x, y: real;
    14            i, n: integer;
    15  begin
    16            for i := 0 to lim do begin
    17                      x := d / i;
    18                      y := exp(-x) * sin(c * x);
    19                      n := round(s * y) + h;
    20                      repeat
    21                                write(' ');
    22                                n := n - 1
    23                      until n = 0;
    24                      writeln('*')
    25            end
    26  end.
tutorial%
```

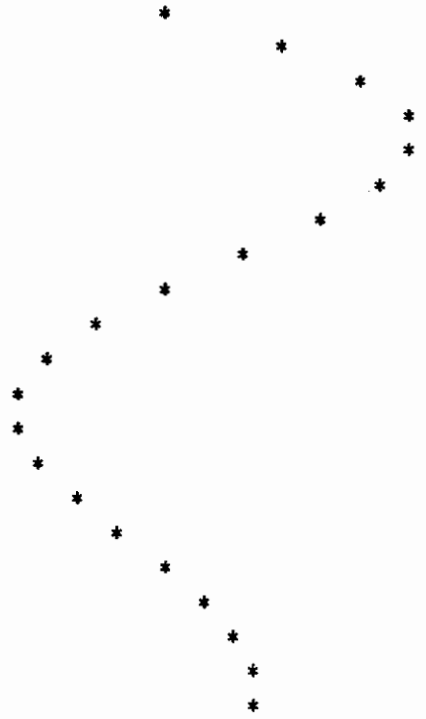
1.4. Executing the Second Example

We are now ready to execute the second example. The following output was produced by our first run.

```
tutorial% px

Statement count limit of 500000 exceeded

        Error in "graph1" + 7 near line 22.
Execution terminated abnormally.

500000 statements executed in 95.68 seconds cpu time.
tutorial%
```

Here the interpreter is presenting us with a runtime error diagnostic. It detected a 'division by zero' at line 17. Examining line 17, we see that we have written the statement 'x := d / i' instead of 'x := d * i'. We can correct this and rerun the program:

```
tutorial% ex bigger.p
"bigger.p" 26 lines, 538 characters
:17
        x := d / i
:s'/'*
        x := d * i
:write
"bigger.p" 26 lines, 538 characters
:q
tutorial% pix bigger.p
```



```
Execution begins...
Execution terminated.

2550 statements executed in 0.67 seconds cpu time.
tutorial%
```

This appears to be the output we wanted. We could now save the output in a file if we wished by using the shell to redirect the output:

```
tutorial% px > graph
```

We can use *cat*(1) to see the contents of the file graph. We can also make a listing of the graph on the line printer without putting it into a file:

```
tutorial% px | lpr
Execution begins...
Execution terminated.

2550 statements executed in 0.70 seconds cpu time.
tutorial%
```

Note here that the statistics lines came out on our terminal. The statistics line comes out on the diagnostic output (unit 2.) There are two ways to get rid of the statistics line. We can redirect the statistics message to the printer using the syntax '|&' to the shell rather than '|', i.e.:

```
tutorial% px |& lpr
tutorial%
```

or we can translate the program with the **p** option disabled on the command line as we did above. This will disable all post-mortem dumping including the statistics line, thus:

```
tutorial% pi -p bigger.p
tutorial% px | lpr
tutorial%
```

This option also disables the statement limit which normally guards against infinite looping. You should not use it until your program is debugged. Also if **p** is specified and an error occurs, you will not get run time diagnostic information to help you determine what the problem is.

1.5. Formatting the Program Listing

It is possible to use special lines within the source text of a program to format the program listing. An empty line (one with no characters on it) corresponds to a 'space' macro in an assembler, leaving a completely blank line without a line number. A line containing only a control-l (form-feed) character will cause a page eject in the listing with the corresponding line number suppressed. This corresponds to an 'eject' pseudo-instruction. --- See also the *Options common to Pi, Pc, and Px* section for details on the **n** and **i** options of *pi*.

1.6. Execution Profiling

An execution profile consists of a structured listing of (all or part of) a program with information about the number of times each statement in the program was executed for a particular run of the program. These profiles can be used for several purposes. In a program which was abnormally terminated due to excessive looping or recursion or by a program fault, the counts can facilitate location of the error. Zero counts mark portions of the program which were not executed; during the early debugging stages they should prompt new test data or a re-examination of the program logic. The profile is perhaps most valuable, however, in drawing attention to the (typically small) portions of the program that dominate execution time. This information can be used for source level optimization.

1.6.1. An Example

A prime number is a number which is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In translating the program we have specified the **-z** option to *pix*. This option causes the translator to generate counters and count instructions sufficient in number to determine the number of times each statement in the program was executed.† When execution of the program completes, either normally or abnormally, this count data is written to the file *pmon.out* in the current directory.‡ It is then possible to prepare an execution profile by giving *pxp* the name of the file associated with this data, as was done in the following example.

```
tutorial% pix -l -z primes.p
Berkeley Pascal PI — Version 2.13 (4/7/83)

Wed Nov  9 15:53 1983  primes.p

    1  program primes(output);
    2  const n = 50; n1 = 7; (*n1 = sqrt(n)*)
    3  var i,k,x,inc,lim,square,l: integer;
    4      prim: boolean;
    5      p,v: array[1..n1] of integer;
```

†The counts are completely accurate only in the absence of runtime errors and nonlocal goto statements. This is not generally a problem, however, as in structured programs nonlocal goto statements occur infrequently, and counts are incorrect after abnormal termination only when the *upward look* described below to get a count passes a suspended call point.

‡*Pmon.out* has a name similar to *mon.out* the monitor file produced by the profiling facility of the C compiler *cc*(1). See *prof*(1) for a discussion of the C compiler profiling facilities.

```
 6  begin
 7      write(2:6, 3:6); l := 2;
 8      x := 1; inc := 4; lim := 1; square := 9;
 9      for i := 3 to n do
10      begin (*find next prime*)
11          repeat x := x + inc; inc := 6-inc;
12              if square <= x then
13                  begin lim := lim+ 1;
14                      v[lim] := square; square := sqr(p[lim+ 1])
15                  end ;
16              k := 2; prim := true;
17              while prim and (k<lim) do
18              begin k := k+ 1;
19                  if v[k] < x then v[k] := v[k] + 2*p[k];
20                  prim := x <> v[k]
21              end
22          until prim;
23          if i <= n1 then p[i] := x;
24          write(x:6); l := l+ 1;
25          if l = 10 then
26              begin writeln; l := 0
27              end
28      end ;
29      writeln;
30  end .
 2     3     5     7    11    13    17    19    23    29
31    37    41    43    47    53    59    61    67    71
73    79    83    89    97   101   103   107   109   113
127   131   137   139   149   151   157   163   167   173
179   181   191   193   197   199   211   223   227   229

Execution begins...
Execution terminated.

1404 statements executed in 0.29 seconds cpu time.
tutorial%
```

1.6.2. Discussion

The header lines of the outputs of *pix* and *pxp* in this example indicate the version of the translator and execution profiler in use at the time this example was prepared. The time given with the file name (also on the header line) indicates the time of last modification of the program source file. This time serves to *version stamp* the input program. *Pxp* also indicates the time at which the profile data was gathered.

```
tutorial% pxp -z primes.p
Berkeley Pascal PXP -- Version 2.11 (2/6/83)

Wed Nov  9 15:53 1983  primes.p
```

```
Profiled Fri Jan 13 09:23 1984

     1          1.-----|program primes(output);
     2                 |const
     2                 |   n = 50;
     2                 |   n1 = 7; (*n1 = sqrt(n)*)
     3                 |var
     3                 |   i, k, x, inc, lim, square, l: integer;
     4                 |   prim: boolean;
     5                 |   p, v: array [1..n1] of integer;
     6                 |begin
     7                 |   write(2: 6, 3: 6);
     7                 |   l := 2;
     8                 |   x := 1;
     8                 |   inc := 4;
     8                 |   lim := 1;
     8                 |   square := 9;
     9                 |   for i := 3 to n do begin (*find next prime*)
     9              48.-----|   repeat
    11                  76.-----|   x := x + inc;
    11                          |   inc := 6 - inc;
    12                          |   if square <= x then begin
    13                       5.-----|   lim := lim + 1;
    14                          |   v[lim] := square;
    14                          |   square := sqr(p[lim + 1])
    14                      |   end;
    16                      |   k := 2;
    16                      |   prim := true;
    17                      |   while prim and (k < lim) do begin
    18                   157.-----|   k := k + 1;
    19                          |   if v[k] < x then
    19                       42.-----|   v[k] := v[k] + 2 * p[k];
    20                          |   prim := x <> v[k]
    20                      |   end
    20                      |until prim;
    23                  |   if i <= n1 then
    23                  5.-----|   p[i] := x;
    24                   |   write(x: 6);
    24                   |   l := l + 1;
    25                   |   if l = 10 then begin
    26                  5.-----|   writeln;
    26                          |   l := 0
    26                   |   end
    26              |   end;
    29              |   writeln
    29              |end.
tutorial%
```

To determine the number of times a statement was executed, one looks to the left of the statement and finds the corresponding vertical bar '|'. If this vertical bar is labelled with a count then that count gives the number of times the statement was executed. If the bar is not

labelled, we look up in the listing to find the first '|' which directly above the original one which has a count and that is the answer. Thus, in our example, *k* was incremented 157 times on line 18, while the *write* procedure call on line 24 was executed 48 times as given by the count on the **repeat**.

More information on *pxp* can be found in its manual section *pxp* (1) and in the *Options available in Px, Options available in Pc,* and *Separate Compilation with Pc* sections below.

Chapter 2

Error diagnostics

This section of the *User's Manual* discusses the error diagnostics of the programs *pi*, *pc* and *px*. *Pix* is a simple but useful program which invokes *pi* and *px* to do all the real processing. See its manual section *pix* (1) and the *Options common to Pi, Pc, and Px* section below for more details. All the diagnostics given by *pi* will also be given by *pc*.

2.1. Translator syntax errors

A few comments on the general nature of the syntax errors usually made by Pascal programmers and the recovery mechanisms of the current translator may help in using the system.

2.1.1. Illegal characters

Characters such as '$', '!', and '@' are not part of the language Pascal. If they are found in the source program, and are not part of a constant string, a constant character, or a comment, they are considered to be 'illegal characters'. This can happen if you leave off an opening string quote '''. Note that the character '"', although used in English to quote strings, is not used to quote strings in Pascal. Most non-printing characters in your input are also illegal except in character constants and character strings. Except for the tab and form feed characters, which are used to ease formatting of the program, non-printing characters in the input file print as the character '?' so that they will show in your listing.

2.1.2. String errors

There is no character string of length 0 in Pascal. Consequently the input '''' is not acceptable. Similarly, encountering an end-of-line after an opening string quote ''' without encountering the matching closing quote yields the diagnostic "Unmatched ' for string". It is permissible to use the character '#' instead of ''' to delimit character and constant strings for portability reasons. For this reason, a spuriously placed '#' sometimes causes the diagnostic about unbalanced quotes. Similarly, a '#' in column one is used when preparing programs which are to be kept in multiple files.

2.1.3. Comments in a comment, non-terminated comments

As we saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without generating this diagnostic since there are two different kinds of comments - those delimited by '{' and '}', and those delimited by '(*' and '*)'. Thus consider:

```
{ This is a comment enclosing a piece of program
a := functioncall;      (* comment within comment *)
procedurecall;
lhs := rhs;         (* another comment *)
}
```

By using one kind of comment exclusively in your program you can use the other delimiters when you need to "comment out" parts of your program. In this way you will also allow the translator to help by detecting statements accidentally placed within comments.

If a comment does not terminate before the end of the input file, the translator will point to the beginning of the comment, indicating that the comment is not terminated. In this case processing will terminate immediately. See the discussion of "QUIT" below.

2.1.4. Digits in numbers

This part of the language is a minor nuisance. Pascal requires digits in real numbers both before and after the decimal point. Thus the following statements, which look quite reasonable to FORTRAN users, generate diagnostics in Pascal:

```
Wed Nov  9 15:53 1983  digits.p:
     4  r := 0.;
e  ------------^---- Digits required after decimal point
     5  r := .0;
e  -----------^---- Digits required before decimal point
     6  r := 1.e10;
e  ------------^---- Digits required after decimal point
     7  r := .05e-10;
e  -----------^---- Digits required before decimal point
```

These same constructs are also illegal as input to the Pascal interpreter *px*.

2.1.5. Replacements, insertions, and deletions

When a syntax error is encountered in the input text, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and considers a set of simple corrections to see whether they will allow the analysis to continue. These corrections involve replacing an input token with a different token, inserting a token, or replacing an input token with a different token. Most of these changes will not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in this case the recovery makes no attempt to determine *which* identifier or *what* number should be inserted, hence these are considered fatal syntax errors.

Consider the following example.

```
tutorial% pix -l synerr.p
Berkeley Pascal PI -- Version 2.13 (4/7/83)

Wed Nov  9 15:52 1983  synerr.p

    1  program syn(output);
    2  var i, j are integer;
e  ---------------↑--- Replaced identifier with a ':'
    3  begin
    4          for j :* 1 to 20 begin
e  --------------------↑--- Replaced '*' with a '='
e  ------------------------------↑--- Inserted keyword do
    5                  write(j);
    6                  i = 2 ** j;
e  ----------------------↑--- Inserted ':'
E  ------------------------------↑--- Inserted identifier
    7                  writeln(i))
E  ------------------------------↑--- Deleted ')'
    8          end
    9  end.
tutorial%
```

The only surprise here may be that Pascal does not have an exponentiation operator, hence the complaint about '**'. This error illustrates that, if you assume that the language has a feature which it does not, the translator diagnostic may not indicate this, as the translator is unlikely to recognize the construct you supply.

2.1.6. Undefined or improper identifiers

If an identifier is encountered in the input but is undefined, the error recovery will replace it with an identifier of the appropriate class. Further references to this identifier will be summarized at the end of the containing **procedure** or **function** or at the end of the **program** if the reference occurred in the main program. Similarly, if an identifier is used in an inappropriate way, e.g. if a **type** identifier is used in an assignment statement, or if a simple variable is used where a **record** variable is required, a diagnostic will be produced and an identifier of the appropriate type inserted. Further incorrect references to this identifier will be flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undefined variable uses are.

2.1.7. Expected symbols, malformed constructs

If none of the above mentioned corrections appear reasonable, the error recovery will examine the input to the left of the point of error to see if there is only one symbol which can follow this input. If this is the case, the recovery will print a diagnostic which indicates that the given symbol was 'Expected'.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates that the input is "malformed". If necessary, the translator may

then skip forward in the input to a place where analysis can continue. This process may cause some errors in the text to be missed.

Consider the following example:

```
tutorial% pix -l synerr2.p
Berkeley Pascal PI — Version 2.13 (4/7/83)

Wed Nov  9 15:52 1983  synerr2.p

    1  program synerr2(input,outpu);
    2  integer a(10)
E -----↑---- Malformed declaration
    3  begin
    4          read(b);
E ------------------↑---- Undefined variable
    5          for c := 1 to 10 do
E ----------------↑---- Undefined variable
    6                  a(c) := b * c;
E --------------------↑---- Undefined procedure
E ------------------------↑---- Malformed statement
    7  end.
E 1 - File outpu listed in program statement but not declared
In program synerr2:
  E - a undefined on line 6
  E - b undefined on lines 4
  E - c undefined on lines 5 6
Execution suppressed due to compilation errors
tutorial%
```

Here we misspelled *output* and gave a FORTRAN style variable declaration which the translator diagnosed as a 'Malformed declaration'. When, on line 6, we used '(' and ')' for subscripting (as in FORTRAN) rather than the '[' and ']' which are used in Pascal, the translator noted that *a* was not defined as a **procedure**. This occurred because **procedure** and **function** argument lists are delimited by parentheses in Pascal. As it is not permissible to assign to procedure calls the translator diagnosed a malformed statement at the point of assignment.

2.1.8. Expected and unexpected end-of-file, "QUIT"

If the translator finds a complete program, but there is more non-comment text in the input file, then it will indicate that an end-of-file was expected. This situation may occur after a bracketing error, or if too many **end**s are present in the input. The message may appear after the recovery says that it "Expected `.'" since '.' is the symbol that terminates a program.

If severe errors in the input prohibit further processing the translator may produce a diagnostic followed by "QUIT". One example of this was given above - a non-terminated comment; another example is a line which is longer than 160 characters. Consider also the following example.

```
tutorial% pix -l mism.p
Berkeley Pascal PI — Version 2.13 (4/7/83)
```

```
Wed Nov  9 15:53 1983  mism.p

    1  program mismatch(output)
    2  begin
e  -----↑---- Inserted ';'
    3          writeln('***');
    4          { The next line is the last line in the file }
    5          writeln
E  --------------------↑---- Malformed declaration
   --------------------↑---- Unexpected end-of-file - QUIT
tutorial%
```

2.2. Translator semantic errors

The extremely large number of semantic diagnostic messages which the translator produces make it unreasonable to discuss each message or group of messages in detail. The messages are, however, very informative. We will here explain the typical formats and the terminology used in the error messages so that you will be able to make sense out of them. In any case in which a diagnostic is not completely comprehensible you can refer to the *User Manual* by Jensen and Wirth for examples.

2.2.1. Format of the error diagnostics

As we saw in the example program above, the error diagnostics from the Pascal translator include the number of a line in the text of the program as well as the text of the error message. While this number is most often the line where the error occurred, it is occasionally the number of a line containing a bracketing keyword like **end** or **until**. In this case, the diagnostic may refer to the previous statement. This occurs because of the method the translator uses for sampling line numbers. The absence of a trailing ';' in the previous statement causes the line number corresponding to the **end** or **until**. to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary to some users. This is the only notable exception, however, to reasonable associations.

2.2.2. Incompatible types

Since Pascal is a strongly typed language, many semantic errors manifest themselves as type errors. These are called 'type clashes' by the translator. The types allowed for various operators in the language are summarized on page 108 of the Jensen-Wirth *User Manual*. It is important to know that the Pascal translator, in its diagnostics, distinguishes between the following type 'classes':

array	Boolean	char	file	integer
pointer	real	record	scalar	string

These words are plugged into a great number of error messages. Thus, if you tried to assign an *integer* value to a *char* variable you would receive a diagnostic like the following:

```
Wed Nov  9 15:53 1983  clash.p:
E 7 - Type clash: integer is incompatible with char
  ... Type of expression clashed with type of variable in assignment
```

In this case, one error produced a two line error message. If the same error occurs more than once, the same explanatory diagnostic will be given each time.

2.2.3. Scalar

The only class whose meaning is not self-explanatory is 'scalar'. Scalar has a precise meaning in the Jensen-Wirth *User Manual* where, in fact, it refers to *char*, *integer*, *real*, and *Boolean* types as well as the enumerated types. For the purposes of the Pascal translator, scalar in an error message refers to a user-defined, enumerated type, such as *ops* in the example above or *color* in

```
type color = (red, green, blue)
```

For integers, the more explicit denotation *integer* is used. Although it would be correct, in the context of the *User Manual* to refer to an integer variable as a *scalar* variable *pi* prefers the more specific identification.

2.2.4. Function and procedure type errors

For built-in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message similar to:

```
Wed Nov  9 15:52 1983  sin1.p:
E 12 - sin takes exactly one argument
```

is given. If the type of the argument is wrong, a message like

```
Wed Nov  9 15:52 1983  sin2.p:
E 12 - sin's argument must be integer or real, not char
```

is produced.

2.2.5. Can't read and write scalars, etc.

The messages which state that scalar (user-defined) types cannot be written to and from files are often mysterious. It is in fact the case that if you define

```
type color = (red, green, blue)
```

"standard" Pascal does not associate these constants with the strings 'red', 'green', and 'blue' in any way. An extension has been added which allows enumerated types to be read and written, however if the program is to be portable, you will have to write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers and real

numbers from text files. You cannot read strings or Booleans. It is possible to make a

```
file of color
```

but the representation is binary rather than string.

2.2.6. Expression diagnostics

The diagnostics for semantically ill-formed expressions are very explicit. Consider this sample translation:

```
tutorial% pi -l expr.p
Berkeley Pascal PI -- Version 2.13 (4/7/83)

Wed Nov  9 15:53 1983  expr.p

    1  program x(output);
    2  var
    3          a: set of char;
    4          b: Boolean;
    5          c: (red, green, blue);
    6          p: ↑ integer;
    7          A: alfa;
    8          B: packed array [1..5] of char;
    9  begin
   10          b := true;
   11          c := red;
   12          new(p);
   13          a := [];
   14          A := 'Hello, yellow';
   15          b := a and b;
   16          a := a * 3;
   17          if input < 2 then writeln('boo');
   18          if p <= 2 then writeln('sure nuff');
   19          if A = B then writeln('same');
   20          if c = true then writeln('hue''s and color''s')
   21  end.
E 14 - Constant string too long
E 15 - Left operand of and must be Boolean, not set
E 16 - Cannot mix sets with integers and reals as operands of *
E 17 - files may not participate in comparisons
E 18 - pointers and integers cannot be compared - operator was <=
E 19 - Strings not same length in = comparison
E 20 - scalars and Booleans cannot be compared - operator was =
e 21 - Input is used but not defined in the program statement
In program x:
  w - constant green is never used
  w - constant blue is never used
  w - variable B is used but never set
tutorial%
```

2.2.9. Goto's into structured statements

The translator detects and complains about **goto** statements which transfer control into structured statements (**for**, **while**, etc.) It does not allow such jumps, nor does it allow branching from the **then** part of an **if** statement into the **else** part. Such checks are made only within the body of a single procedure or function.

2.2.10. Unused variables, never set variables

Although *pi* always clears variables to 0 at **procedure** and **function** entry, *pc* does not unless runtime checking is enabled using the **C** option. It is **not** good programming practice to rely on this initialization. To discourage this practice, and to help detect errors in program logic, *pi* flags as a 'w' warning error:

1) Use of a variable which is never assigned a value.
2) A variable which is declared but never used, distinguishing between those variables for which values are computed but which are never used, and those completely unused.

In fact, these diagnostics are applied to all declared items. Thus a **const** or a **procedure** which is declared but never used is flagged. The **w** option of *pi* may be used to suppress these warnings; see the *Options* and *Options common to Pi, Pc, and Px* sections, below.

2.3. Translator panics, i/o errors

2.3.1. Panics

One class of error which rarely occurs, but which causes termination of all processing when it does is a panic. A panic indicates a translator-detected internal inconsistency. A typical panic message is:

```
snark (rvalue) line=110 yyline=109
Snark in pi
```

If you receive such a message, the translation will be quickly and perhaps ungracefully terminated. Contact your system manager after saving a copy of your program for later inspection. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by ascertaining which change caused the *snark* and making a different change or correcting an error in the program. A small number of panics are possible in *px*.

2.3.2. Out of memory

The only other error which will abort translation when no errors are detected is running out of memory. All tables in the translator, with the exception of the parse stack, are dynamically allocated, and can grow to take up a good deal of process space. In general, very large programs should be translated using *pc* and the separate compilation facility.

If you receive an out of space message from the translator during translation of a large **procedure** or **function** or one containing a large number of string constants you may yet be able

to translate your program if you break this one **procedure** or **function** into several routines.

2.3.3. I/O errors

Other errors which you may encounter when running *pi* relate to input-output. If *pi* cannot open the file you specify, or if the file is empty, you will be so informed.

2.4. Run-time errors

We saw, in our second example, a run-time error. We here give the general description of run-time errors. The more unusual interpreter error messages are explained briefly in the manual section for *px* (1).

2.4.1. Start-up errors

These errors occur when the object file to be executed is not available or appropriate. Typical errors here are caused by the specified object file not existing, not being a Pascal object, or being inaccessible to the user.

2.4.2. Program execution errors

These errors occur when the program interacts with the Pascal runtime environment in an inappropriate way. Typical errors are values or subscripts out of range, bad arguments to built-in functions, exceeding the statement limit because of an infinite loop, or running out of memory‡. The interpreter will produce a backtrace after the error occurs, showing all the active routine calls, unless the **p** option was disabled when the program was translated. Unfortunately, no variable values are given and no way of extracting them is available.*

As an example of such an error, assume that we have accidentally declared the constant *n1* to be 6, instead of 7 on line 2 of the program primes as given in the *Execution profiling* section above. If we run this program we get the following response.

```
tutorial% pix primes.p
    2     3     5     7    11    13    17    19    23    29
   31    37    41    43    47    53    59    61    67    71
   73    79    83    89    97   101   103   107   109   113
  127   131   137   139   149   151   157   163   167Execution begins...

Subscript value of 7 is out of range

        Error in "primes" + 8 near line 14.
Execution terminated abnormally.
```

‡The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a core image when it runs out of memory. This situation occurs very rarely.

* On the VAX-11, each variable is restricted to allocate at most 65000 bytes of storage (this is a PDP-11ism that has survived to the VAX.)

```
941 statements executed in 0.21 seconds cpu time.
tutorial%
```

Here the interpreter indicates that the program terminated abnormally due to a subscript out of range near line 14, which is eight lines into the body of the program primes.

2.4.3. Interrupts

If the program is interrupted while executing and the p option was not specified, then a backtrace will be printed.† The file *pmon.out* of profile information will be written if the program was translated with the s option enabled to *pi* or *pix*.

2.4.4. I/O interaction errors

The final class of interpreter errors results from inappropriate interactions with files, including the user's terminal. Included here are bad formats for integer and real numbers (such as no digits after the decimal point) when reading.

†Occasionally, the Pascal system will be in an inconsistent state when this occurs, e.g. when an interrupt terminates a **procedure** or **function** entry or exit. In this case, the backtrace will only contain the current line. A reverse call order list of procedures will not be given.

Chapter 3

Data Representations

This chapter describes the ways that FORTRAN-77 represents data in storage. This chapter is intended as a guide to those programmers who wish to write modules in languages other than FORTRAN-77 and have those modules interface to FORTRAN-77.

3.1. Storage Allocation

This section describes the way in which storage is allocated to variables of various types.

In general, any *word* value (a value which occupies 16 bits) is always aligned on a word boundary. Anything larger than a word is also aligned on a word boundary. Values that can fit into a single byte are aligned on a byte boundary.

integer*2
occupies 16 bits (two bytes or one word), aligned on a word boundary.

integer and **integer*4**
occupy 32 bits (four bytes or two words), aligned on a word boundary.

real and **real*4**
occupy 32 bits (four bytes or two words), aligned on a word boundary. A **real** element has a sign bit, an 8-bit exponent and a 23-bit mantissa. FORTRAN-77 **real** elements conform to the IEEE standard for reals as defined in the March 1981 Computer magazine. The layout of a **real** element is shown below.

double precision and **real*8**
elements occupy 64 bits (eight bytes or four words), aligned on a word boundary. A **double precision** element has a sign bit, an 11-bit exponent and a 52-bit mantissa. FORTRAN-77 **double precision** elements conform to the IEEE standard for double precision data as defined in the March 1981 Computer magazine. The layout of a **double precision** element is shown below.

complex
elements are represented by two **real** elements. The first element represents the real part of the number, the second represents the imaginary part.

double complex
elements are represented by two **double precision** elements. The first element represents the real part of the number, the second represents the imaginary part.

logical*2
occupies two bytes (16 bits) of storage, aligned on a word boundary. A value of 0 represents the value **.false.** . A value of 1 represents the value **.true.** . Any other value is an 'undefined' logical value.

logical and **logical*4**
occupies four bytes (32 bits) of storage, aligned on a word boundary. A value of 0 represents the value **.false.** . A value of 1 represents the value **.true.** . Any other value is an 'undefined' logical value.

3.2. Data Representations

Whatever the size of the data element in question, the most significant bit of the data element is always in the lowest numbered byte of however many bytes are required to represent that object.

3.2.1. Representation of real and double precision

real and **double precision** data elements are represented according to the proposed IEEE standard described in Computer magazine of March, 1981:

Table 3-1: Representation of Real and Double Precision Numbers

	Single Precision	*Double Precision*
Sign	bit 31	bit 63
Exponent	bits 30–23 bias 127	bits 62–52 bias 1023
Mantissa	bits 22–0	bits 51–0

The parts of **real** and **double precision** numbers are as follows:

- a one-bit sign bit. The sign bit is a **1** if, and only if, the number is negative.
- a biased exponent. The exponent is eight bits for a **real** number, and is eleven bits for a **double precision** number. The values of all zeros, and all ones, are reserved values for exponents.
- a normalized mantissa, with the high-order **1** bit 'hidden'. The mantissa is 23 bits for a **real** number, and is 52 bits for a **double precision** number. A **real** or **double precision** number is represented by the form:

  $2^{exponent-bias} * 1.f$

  where 'f' is the bits in the mantissa.

3.2.2. Representation of Extreme Numbers

zero (signed)
: is represented by an exponent of zero, and a mantissa of zero.

denormalized numbers
: are a product of 'gradual underflow'. They are non-zero numbers with an exponent of zero. The form of a denormalized number is:

  $2^{exponent-bias+1} * 0.f$

  where 'f' is the bits in the mantissa.

signed infinity
: (that is, affine infinity) is represented by the largest value that the exponent can assume (all

problem even with this simple example. Some Pascal systems attach meaning to the ordering of the file in the **program** statement file list. Berkeley Pascal does not do so.

3.2. Eof and eoln

An extremely common problem encountered by new users of Pascal, especially in the interactive environment offered by UNIX, relates to the definitions of *eof* and *eoln*. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line or the end of a file. Setting *eof* or *eoln* actually corresponds to an implicit read in which the input is inspected, but no input is "used up". In fact, there is no way the system can know whether the input is at the end-of-file or the end-of-line unless it attempts to read a line from it. If the input is from a previously created file, then this reading can take place without run-time action by the user. However, if the input is from a terminal, then the input is what the user types.† If the system were to do an initial read automatically at the beginning of program execution, and if the input were a terminal, the user would have to type some input before execution could begin. This would make it impossible for the program to begin by prompting for input or printing a herald.

Berkeley Pascal has been designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end-of-file or end-of-line conditions are true. Thus, internally, these functions can have three values – true, false, and "I don't know yet; if you ask me I'll have to find out". All files remain in this last, indeterminate state until the Pascal program requires a value for *eof* or *eoln* either explicitly or implicitly, e.g. in a call to *read*. The important point to note here is that if you force the Pascal system to determine whether the input is at the end-of-file or the end-of-line, it will be necessary for it to attempt to read from the input.

Thus consider the following example code

```
while not eof do begin
    write('number, please? ');
    read(i);
    writeln('that was a ', i: 2)
end
```

At first glance, this may be appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the **while** loop asks whether *eof* is true *before* the request is printed. This will force the Pascal system to decide whether the input is at the end-of-file. The Pascal system will give no messages; it will simply wait for the user to type a line. By producing the desired prompting before testing *eof,* the following code avoids this problem:

```
write('number, please ? ');
while not eof do begin
    read(i);
    writeln('that was a ', i:2);
    write('number, please ? ')
end
```

The user must still type a line before the **while** test is completed, but the prompt will ask for

†It is not possible to determine whether the input is a terminal, as the input may appear to be a file but actually be a *pipe,* the output of a program which is reading from the terminal.

it. This example, however, is still not correct. To understand why, it is first necessary to know, as we will discuss below, that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or real numbers, is defined so that, if there are only blanks left in the file, it will return a zero value and set the end-of-file condition. If, however, there is a number remaining in the file, the end-of-file condition will not be set even if it is the last number, as *read* never reads the blanks after the number, and there is always at least one blank. Thus the modified code will still put out a spurious

```
that was a 0
```

at the end of a session with it when the end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure *readln* instead of *read* here. In general, unless we test the end-of-file condition both before and after calls to *read* or *readln*, there will be inputs for which our program will attempt to read past end-of-file.

3.3. More about eoln

To have a good understanding of when *eoln* will be true it is necessary to know that in any file there is a special character indicating end-of-line, and that, in effect, the Pascal system always reads one character ahead of the Pascal *read* commands.† For instance, in response to 'read(ch)', the system sets *ch* to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the new-line character, the normal UNIX line separator. When the read routine gets the new-line character, it replaces that character by a blank (causing every line to end with a blank) and sets *eoln* to true. *Eoln* will be true as soon as we read the last character of the line and **before** we read the blank character corresponding to the end of line. Thus it is almost always a mistake to write a program which deals with input in the following way:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

as this will almost surely have the effect of ignoring the last character in the line. The 'read(ch)' belongs as part of the normal processing.

Given this framework, it is not hard to explain the function of a *readln* call, which is defined as:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end-of-line is the current input symbol and then discards this blank. The next character available from *read* will therefore be the first character of the next line, if one exists.

†In Pascal terms, 'read(ch)' corresponds to 'ch := input^; get(input)'

3.4. Output buffering

A final point about Pascal input-output must be noted here. This concerns the buffering of the file *output*. It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output; even less efficient if the output is the input of another program such as the line printer daemon *lpr* (1). To gain efficiency, the Pascal system "buffers" the output characters (i.e. it saves them in memory until the buffer is full and then emits the entire buffer in one system interaction.) However, to allow interactive prompting to work as in the example given above, this prompt must be printed before the Pascal system waits for a response. For this reason, Pascal normally prints all the output which has been generated for the file *output* whenever

1) A *writeln* occurs, or
2) The program reads from the terminal, or
3) The procedure *message* or *flush* is called.

Thus, in the code sequence

```
for i := 1 to 5 do begin
    write(i: 2);
    Compute a lot with no output
end;
writeln
```

the output integers will not print until the *writeln* occurs. The delay can be somewhat disconcerting, and you should be aware that it will occur. By setting the b option to 0 before the **program** statement by inserting a comment of the form

```
(*$b0*)
```

we can cause *output* to be completely unbuffered, with a corresponding horrendous degradation in program efficiency. Option control in comments is discussed in the *Options* section below.

3.5. Files, reset, and rewrite

It is possible to use extended forms of the built-in functions *reset* and *rewrite* to get more general associations of UNIX file names with Pascal file variables. When a file other than *input* or *output* is to be read or written, then the reading or writing must be preceded by a *reset* or *rewrite* call. In general, if the Pascal file variable has never been used before, there will be no UNIX filename associated with it. By mentioning the file in a **program** statement, however, we can cause a UNIX file with the same name as the Pascal variable to be associated with it. If we do not mention a file in the **program** statement and use it for the first time with the statement

```
reset(f)
```

or

```
rewrite(f)
```

then the Pascal system will generate a temporary name of the form 'tmp.x' for some character 'x', and associate this UNIX file name name with the Pascal file. The first such generated name will be 'tmp.1' and the names continue by incrementing their last character through the ASCII set. The advantage of using such temporary files is that they are automatically *remove*d by the

Pascal system as soon as they become inaccessible. They are not removed, however, if a run-time error causes termination while they are in scope.

To cause a particular UNIX pathname to be associated with a Pascal file variable we can give that name in the *reset* or *rewrite* call, e.g. we could have associated the Pascal file *data* with the file 'primes' in our example in the *Translator syntax errors* section above by doing:

```
reset(data, 'primes')
```

instead of a simple

```
reset(data)
```

In this case it is not essential to mention 'data' in the program statement, but it is still a good idea because is serves as an aid to program documentation. The second parameter to *reset* and *rewrite* may be any string value, including a variable. Thus the names of UNIX files to be associated with Pascal file variables can be read in at run time. Full details on file name/file variable associations are given in *Restriction and limitations* section of the appendix.

3.6. Argc and argv

Each UNIX process receives a variable length sequence of arguments each of which is a variable length character string. The built-in function *argc* and the built-in procedure *argv* can be used to access and process these arguments. The value of the function *argc* is the number of arguments to the process. By convention, the arguments are treated as an array, and indexed from 0 to *argc*−1, with the zeroth argument being the name of the program being executed. The rest of the arguments are those passed to the command on the command line. Thus, the command

```
tutorial% obj /etc/motd /usr/dict/words hello
```

will invoke the program in the file *obj* with *argc* having a value of 4. The zeroth element accessed by *argv* will be 'obj', the first '/etc/motd', etc.

Pascal does not provide variable size arrays, nor does it allow character strings of varying length. For this reason, *argv* is a procedure and has the syntax

```
argv(i, a)
```

where *i* is an integer and *a* is a string variable. This procedure call assigns the (possibly truncated or blank padded) *i* 'th argument of the current process to the string variable *a*. The file manipulation routines *reset* and *rewrite* will strip trailing blanks from their optional second arguments so that this blank padding is not a problem in the usual case where the arguments are file names.

We are now ready to give a Berkeley Pascal program 'kat', based on that given in the *Translator syntax errors* section above, which can be used with the same syntax as the UNIX system program *cat* (1).

```
tutorial% cat kat.p
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
```

```
begin
   i := 1;
   repeat
      if i < argc then begin
         argv(i, name);
         reset(input, name);
         i := i + 1
      end;
      while not eof do begin
         while not eoln do begin
            read(ch);
            write(ch)
         end;
         readln;
         writeln
      end
   until i >= argc
end { kat }.
tutorial%
```

Note that the *reset* call to the file *input* here, which is necessary for a clear program, may be disallowed on other systems. As this program deals mostly with *argc* and *argv* and UNIX system dependent considerations, portability is of little concern.

If this program is in the file 'kat.p', then we can do

```
tutorial% pi kat.p
tutorial% mv obj kat
tutorial% kat primes
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229

930 statements executed in 0.16 seconds cpu time.
tutorial% kat
This is a line of text.
This is a line of text.
The next line contains only an end-of-file (an invisible control-d!)
The next line contains only an end-of-file (an invisible control-d!)

287 statements executed in 0.04 seconds cpu time.
tutorial%
```

Thus we see that, if it is given arguments, 'kat' will, like *cat,* copy each one in turn. If no arguments are given, it copies from the standard input. Thus it will work as it did before, with

```
tutorial% kat < primes
```

now equivalent to

```
tutorial% kat primes
```

although the mechanisms are quite different in the two cases. Note that if 'kat' is given a bad file name, for example:

```
tutorial% kat xxxxqqq

Could not open xxxxqqq: No such file or directory

    Error in "kat"+ 5 near line 11.

4 statements executed in 0.00 seconds cpu time.
tutorial%
```

it will give a diagnostic and a post-mortem control flow backtrace for debugging. If we were going to use 'kat', we might want to translate it differently, e.g.:

```
tutorial% pi -pb kat.p
tutorial% mv obj kat
```

Here we have disabled the post-mortem statistics printing, so as not to get the statistics or the full traceback on error. The **b** option will cause the system to block buffer the input/output so that the program will run more efficiently on large files. We could have also specified the **t** option to turn off runtime tests if that was felt to be a speed hindrance to the program. Thus we can try the last examples again:

```
tutorial% kat xxxxqqq

Could not open xxxxqqq: No such file or directory

    Error in "kat"
tutorial% kat primes
   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97  101  103  107  109  113
 127  131  137  139  149  151  157  163  167  173
 179  181  191  193  197  199  211  223  227  229

tutorial%
```

The interested reader may wish to try writing a program which accepts command line arguments like *pi* does, using *argc* and *argv* to process them.

Chapter 4

Details on the Components of the System

4.1. Options

The programs *pi*, *pc*, and *pzp* take a number of options.† There is a standard UNIX convention for passing options to programs on the command line, and this convention is followed by the Berkeley Pascal system programs. As we saw in the examples above, option related arguments consisted of the character '–' followed by a single character option name.

Except for the **b** option which takes a single digit value, each option may be set on (enabled) or off (disabled.) When an on/off valued option appears on the command line of *pi* or it inverts the default setting of that option. Thus

```
tutorial% pi -l foo.p
```

enables the listing option **l**, since it defaults off, while

```
tutorial% pi -t foo.p
```

disables the run time tests option **t**, since it defaults on.

In additon to inverting the default settings of *pi* options on the command line, it is also possible to control the *pi* options within the body of the program by using comments of a special form illustrated by

```
{$l-}
```

Here we see that the opening comment delimiter (which could also be a '(*') is immediately followed by the character '$'. After this '$', which signals the start of the option list, we can place a sequence of letters and option controls, separated by ',' characters. The most basic actions for options are to set them, thus

```
{$l+ Enable listing}
```

or to clear them

```
{$t-,p- No run-time tests, no post mortem analysis}
```

†As *pix* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.

Notice that '+ ' always enables an option and '-' always disables it, no matter what the default is. Thus '-' has a different meaning in an option comment than it has on the command line. As shown in the examples, normal comment text may follow the option list.

4.2. Options Common to Pi, Pc, and Pix

The following options are common to both the compiler and the interpreter. With each option we give its default setting, the setting it would have if it appeared on the command line, and a sample command using the option. Most options are on/off valued, with the **b** option taking a single digit value.

4.2.1. b — Buffering of the File 'output'

The **b** option controls the buffering of the file *output*. The default is line buffering, with flushing at each reference to the file *input* and under certain other circumstances detailed in the *Options available in Pc* section below. Mentioning **b** on the command line, that is:

```
tutorial% pi -b assembler.p
```

makes standard output block-buffered, where a block is some system-defined number of characters. The **b** option may also be controlled in comments. It, unique among the Berkeley Pascal options, takes a single digit value rather than an on or off setting. A value of 0, that is:

```
{$b0}
```

makes *output* unbuffered. Any value 2 or greater causes block buffering and is equivalent to the flag on the command line. The option control comment setting **b** must precede the **program** statement.

4.2.2. i — Include File Listing

The **i** option takes the name of an **include** file, **procedure** or **function** name and causes it to be listed while translating†. Typical uses would be

```
tutorial% pix -i scanner.i compiler.p
```

to make a listing of the routines in the file scanner.i, and

```
tutorial% pix -i scanner compiler.p
```

to make a listing of only the routine *scanner*. This option is especially useful for conservation-minded programmers making partial program listings.

†Include files are discussed in the *Multi-file programs* section below.

4.2.3. l — Make a Listing

The **l** option enables a listing of the program. The **l** option defaults off. When specified on the command line, it causes a header line identifying the version of the translator in use and a line giving the modification time of the file being translated to appear before the actual program listing. The **l** option is pushed and popped by the **i** option at appropriate points in the program.

4.2.4. s — Standard Pascal Only

The **s** option causes many of the features of the UNIX implementation which are not found in standard Pascal to be diagnosed as 's' warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features which are diagnosed are: non-standard **procedures** and **functions**, extensions to the **procedure** *write*, and the padding of constant strings with blanks. In addition, all letters are mapped to lower case except in strings and characters so that the case of keywords and identifiers is effectively ignored. The **s** option is most useful when a program is to be transported, thus

```
tutorial% pi -s isitstd.p
```

produces warnings unless the program meets the standard.

4.2.5. t and C — Runtime Tests

These options control the generation of tests that subrange variable values are within bounds at run time. *pi* defaults to generating tests and uses the option **t** to disable them. *pc* defaults to not generating tests, and uses the option **C** to enable them. Disabling runtime tests also causes **assert** statements to be treated as comments.‡

4.2.6. w — Suppress Warning Diagnostics

The **w** option, which defaults on, allows the translator to print a number of warnings about inconsistencies it finds in the input program. Turning this option off with a comment of the form

```
{$w-}
```

or on the command line

```
tutorial% pi -w tryme.p
```

suppresses these usually useful diagnostics.

4.2.7. z — Generate Counters for a pxp Execution Profile

The **z** option, which defaults off, enables the production of execution profiles. By specifying **z** on the command line:

‡See the section on the **Assert statement** in the appendix for details.

```
tutorial% pi -s foo.p
```

or by enabling it in a comment before the **program** statement causes *pi* and *pc* to insert operations in the interpreter code to count the number of times each statement was executed. An example of using *pxp* was given in the *Execution profiling* section above; its options are described in the *Options available in Pxp* section below. Note that the **s** option cannot be used on separately compiled programs.

4.3. Options available in Pi

4.3.1. p — Post-Mortem Dump

The **p** option defaults on, and causes the runtime system to initiate a post-mortem backtrace when an error occurs. The **p** option also makes *px* count statements in the executing program, enforcing a statement limit to prevent infinite loops. Specifying **p** on the command line disables these checks and the ability to give this post-mortem analysis. It does make smaller and faster programs, however. It is also possible to control the **p** option in comments. To prevent the post-mortem backtrace on error, **p** must be off at the end of the **program** statement. Thus, the Pascal cross-reference program was translated with

```
tutorial% pi -pbt pxref.p
```

4.4. Options available in Px

The first argument to *px* is the name of the file containing the program to be interpreted. If no arguments are given, then the file *obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins *argc* and *argv* as described in the *Argc and argv* section above.

Px may also be invoked automatically. In this case, whenever a Pascal object file name is given as a command, the command will be executed with *px* prepended to it; that is

```
tutorial% obj primes
```

will be converted to read

```
tutorial% px obj primes
```

4.5. Options available in Pc

4.5.1. S — Generate Assembly Language

The program is compiled and the assembly language output is left in file appended .s. Thus

```
tutorial% pc -S foo.p
```

creates a file *foo.s*. No executable file is created.

4.5.2. g — Symbolic Debugger Information

The **g** option causes the compiler to generate information needed by *adb*(1), the assembly-level debugger.

4.5.3. o — Redirect the Output File

The *name* argument after the **-o** is used as the name of the output file instead of *a.out*. Its typical use is to name the compiled program using the root of the file name. Thus:

```
tutorial% pc -o myprog myprog.p
```

causes the compiled program to be called *myprog*.

4.5.4. p and pf — Generate Counters for an Execution Profile

The compiler produces code which counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system rather than by inline counters used by *pxp*. This results in less degradation in execution, at somewhat of a loss in accuracy. '- p' causes a *mon.out* file to be produced for *prof*(1) (Q.V.). '-pg' causes a *gmon.out* file to be produced for *gprof*(1)(Q.V.), a more elaborate profiling tool.

4.5.5. O — Run the Object Code Optimizer

The output of the compiler is run through the object code optimizer. This provides an increase in compile time in exchange for a decrease in compiled code size and execution time.

4.6. Options available in Pxp

Pxp takes, on its command line, a list of options followed by the program file name, which must end in '.p' as it must for *pi*, *pc*, and *piz*. *Pxp* will produce an execution profile if any of the **z**, **t** or **c** options is specified on the command line. If none of these options is specified, then *pxp* functions as a program reformatter.

It is important to note that only the **s** and **w** options of *pxp*, which are common to *pi*, *pc*, and *pxp* can be controlled in comments. All other options must be specified on the command line to have any effect.

The following options are relevant to profiling with *pxp*:

4.6.1. a — Include the Bodies of All Routines in the Profile

Pxp normally suppresses printing the bodies of routines which were never executed, to make the profile more compact. This option forces all routine bodies to be printed.

4.6.2. d — Suppress Declaration Parts from a Profile

Normally a profile includes declaration parts. Specifying **d** on the command line suppresses declaration parts.

4.6.3. e — Eliminate include Directives

Normally, *pxp* preserves **include** directives to the output when reformatting a program, as though they were comments. Specifying **–e** causes the contents of the specified files to be reformatted into the output stream instead. This is an easy way to eliminate **include** directives, for example, before transporting a program.

4.6.4. f — Fully Parenthesize Expressions

Normally *pxp* prints expressions with the minimal parenthesization necessary to preserve the structure of the input. This option causes *pxp* to fully parenthesize expressions. Thus the statement which prints as

```
d := a + b mod c / e
```

with minimal parenthesization, the default, will print as

```
d := a + ((b mod c) / e)
```

with the **f** option specified on the command line.

4.6.5. j — Left Justify all Procedures and Functions

Normally, each **procedure** and **function** body is indented to reflect its static nesting depth. This option prevents this nesting and can be used if the indented output would be too wide.

4.6.6. t — Print a Table Summarizing Procedure and Function Calls

The **t** option causes *pxp* to print a table summarizing the number of calls to each **procedure** and **function** in the program. It may be specified in combination with the **s** option, or separately.

4.6.7. z — Enable and Control the Profile

The **z** profile option is very similar to the **i** listing control option of *pi*. If **z** is specified on the command line, then all arguments up to the source file argument which ends in '.p' are taken to be the names of **procedures** and **functions** or **include** files which are to be profiled. If this list is null, then the whole file is to be profiled. A typical command for extracting a profile of part of a large program would be

```
tutorial% pxp -z test parser.i compiler.p
```

This specifies that profiles of the routines in the file *parser.i* and the routine *test* are to be made.

4.7. Formatting programs using pxp

The program *pxp* can be used to reformat programs, by using a command of the form

```
tutorial% pxp dirty.p > clean.p
```

Note that since the shell creates the output file 'clean.p' before *pxp* executes, so 'clean.p' and 'dirty.p' must not be the same file.

Pxp automatically paragraphs the program, performing housekeeping chores such as comment alignment, and treating blank lines, lines containing exactly one blank and lines containing only a form-feed character as though they were comments, preserving their vertical spacing effect in the output. *Pxp* distinguishes between four kinds of comments:

1. Left marginal comments, beginning in the first column of the input line and are placed in the first column of an output line.
2. Aligned comments, preceded by no input tokens on the input line are aligned in the output with the running program text.
3. Trailing comments, preceded in the input line by a token with no more than two spaces separating the token from the comment.
4. Right marginal comments, preceded in the input line by a token from which they are separated by at least three spaces or a tab are aligned down the right margin of the output, currently to the first tab stop after the 40th column from the current "left margin".

Consider the following program.

```
tutorial% cat comments.p
{ This is a left marginal comment. }
program hello(output);
var i : integer; {This is a trailing comment}
j : integer;    {This is a right marginal comment}
k : array [ 1..10] of array [1..10] of integer; {Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
begin
i := 1; {Trailing i comment}
{A left marginal comment}
 {An aligned comment}
j := 1;         {Right marginal comment}
k[1] := 1;
writeln(i, j, k[1])
end.
```

When formatted by *pxp* the following output is produced.

```
tutorial% pxp comments.p
```

```
{ This is a left marginal comment. }

program hello(output);
var
   i: integer; {This is a trailing comment}
   j: integer;                                  {This is a right marginal comment}
   k: array [1..10] of array [1..10] of integer;    {Marginal, but past the margin}
{
 An aligned, multi-line comment
 which explains what this program is
 all about
}
begin
   i := 1; {Trailing i comment}
{A left marginal comment} {An aligned comment}
   j := 1;                                      {Right marginal comment}
   k[1] := 1;
   writeln(i, j, k[1])
end.
tutorial%
```

The following formatting related options are currently available in *pxp*. The options **f** and **j** described in the previous section may also be of interest.

4.7.1. s — Strip Comments

The **s** option causes *pxp* to remove all comments from the input text.

4.7.2. _ — Underline Keywords

A command line argument of the form **-_** as in

```
tutorial% pxp -_ dirty.p
```

can be used to cause *pxp* to underline all keywords in the output for enhanced readability.

4.7.3. [23456789] — Specify Indenting Unit

The normal unit which *pxp* uses to indent a structure statement level is 4 spaces. By giving an argument of the form *-d* with *d* a digit, $2 \leq d \leq 9$ you can specify that *d* spaces are to be used per level instead.

4.8. Pxref

The cross-reference program *pxref* may be used to make cross-referenced listings of Pascal programs. To produce a cross-reference of the program in the file 'foo.p' one can execute the command:

```
tutorial% pxref foo.p
```

The cross-reference is, unfortunately, not block structured. Full details on *pxref* are given in its manual section *pxref* (1).

4.9. Multi-file programs

A text inclusion facility is available with Berkeley Pascal. This facility allows the interpolation of source text from other files into the source stream of the translator. It can be used to divide large programs into more manageable pieces for ease in editing, listing, and maintenance.

The **include** facility is based on that of the UNIX C compiler. To trigger it you can place the character '#' in the first portion of a line and then, after an arbitrary number of blanks or tabs, the word 'include' followed by a filename enclosed in single '' or double "" quotation marks. The file name may be followed by a semicolon ';' if you wish to treat this as a pseudo-Pascal statement. The filenames of included files must end in '.i'. An example of the use of included files in a main program would be:

```
program compiler(input, output, obj);

#include "globals.i"
#include "scanner.i"
#include "parser.i"
#include "semantics.i"

begin
   { main program }
end.
```

At the point the **include** pseudo-statement is encountered in the input, the lines from the included file are interpolated into the input stream. For the purposes of translation and run-time diagnostics and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from 1. Nested includes are possible up to 10 deep.

See the descriptions of the **i** option of *pi* in the *Options common to Pi, Pc, and Pix* section above; this can be used to control listing when **include** files are present.

When a non-trivial line is encountered in the source text after an **include** finishes, the 'popped' filename is printed, in the same manner as above.

For the purposes of error diagnostics when not making a listing, the filename is printed before each diagnostic if the current filename has changed since the last filename was printed.

4.10. Separate Compilation with Pc

A separate compilation facility is provided with the Berkeley Pascal compiler, *pc*. This facility allows programs to be divided into a number of files and the pieces to be compiled individually, to be linked together at some later time. This is especially useful for large programs, where small changes would otherwise require time-consuming re-compilation of the entire program.

Normally, *pc* expects to be given entire Pascal programs. However, if given the –c option on the command line, it will accept a sequence of definitions and declarations, and compile them into a .o file, to be linked with a Pascal program at a later time. In order that procedures and functions be available across separately compiled files, they must be declared with the directive **external**. This directive is similar to the directive **forward** in that it must precede the

resolution of the function or procedure, and formal parameters and function result types must be specified at the **external** declaration and may not be specified at the resolution.

Type checking is performed across separately compiled files. Since Pascal type defintions define unique types, any types which are shared between separately compiled files must be the same definition. This seemingly impossible problem is solved using a facility similar to the **include** facility discussed above. Definitions may be placed in files with the extension **.h** and the files included by separately compiled files. Each definition from a **.h** file defines a unique type, and all uses of a definition from the same **.h** file define the same type. Similarly, the facility is extended to allow the definition of **consts** and the declaration of **labels**, **vars**, and **external functions** and **procedures**. Thus **procedures** and **functions** which are used between separately compiled files must be declared **external**, and must be so declared in a **.h** file included by any file which calls or resolves the **function** or **procedure**. Conversely, **functions** and **procedures** declared **external** may only be so declared in **.h** files. These files may be included only at the outermost level, and thus define or declare global objects. Note that since only **external function** and **procedure** declarations (and not resolutions) are allowed in **.h** files, statically nested **functions** and **procedures** can not be declared **external**.

An example of the use of included **.h** files in a program would be:

```
program compiler(input, output, obj);

#include "globals.h"
#include "scanner.h"
#include "parser.h"
#include "semantics.h"

begin
   { main program }
end.
```

This might include in the main program the definitions and declarations of all the global **labels**, **consts**, **types** **vars** from the file globals.h, and the **external function** and **procedure** declarations for each of the separately compiled files for the scanner, parser and semantics. The header file *scanner.h* would contain declarations of the form:

```
type
   token = record
      { token fields }
   end;

function scan(var inputfile: text): token;
   external;
```

Then the scanner might be in a separately compiled file containing:

```
#include "globals.h"
#include "scanner.h"

function scan;
begin
    { scanner code }
end;
```

which includes the same global definitions and declarations and resolves the scanner functions and procedures declared **external** in the file scanner.h.

Appendix A

Appendix to Jensen and Wirth Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report* and, with that Report, precisely defines the Berkeley implementation. This appendix includes a summary of extensions to the language, gives the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available.

A.1. Extensions to the language Pascal

This section defines non-standard language constructs available in Berkeley Pascal. The **s** standard Pascal option of the translators *pi* and *pc* can be used to detect these extensions in programs which are to be transported.

A.1.1. String padding

Berkeley Pascal will pad constant strings with blanks in expressions and as value parameters to make them as long as is required. The following is a legal Berkeley Pascal program:

```
program x(output);
var z : packed array [ 1 .. 13 ] of char;
begin
   z := 'red';
   writeln(z)
end;
```

The padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := 'red          '
```

which is standard Pascal.

A.1.2. Octal constants, octal and hexadecimal write

Octal constants may be given as a sequence of octal digits followed by the character 'b' or 'B'. The forms

```
write(a:n oct)
```

and

write(a:n **hex**)

cause the internal representation of expression *a*, which must be Boolean, character, integer, pointer, or a user-defined enumerated type, to be written in octal or hexadecimal respectively.

A.1.3. Assert statement

An **assert** statement causes a *Boolean* expression to be evaluated each time the statement is executed. A runtime error results if any of the expressions evaluates to be *false*. The **assert** statement is treated as a comment if run-time tests are disabled. The syntax for **assert** is:

assert <expr>

A.1.4. Enumerated type input-output

Enumerated types may be read and written. On output the string name associated with the enumerated value is output. If the value is out of range, a runtime error occurs. On input an identifier is read and looked up in a table of names associated with the type of the variable, and the appropriate internal value is assigned to the variable being read. If the name is not found in the table a runtime error occurs.

A.1.5. Structure returning functions

An extension has been added which allows functions to return arbitrary sized structures rather than just scalars as in the standard.

A.1.6. Separate compilation

The compiler *pc* has been extended to allow separate compilation of programs. Procedures and functions declared at the global level may be compiled separately. Type checking of calls to separately compiled routines is performed at load time to insure that the program as a whole is consistent. See the section *Separate compilation with Pc*, above, for details.

A.2. Resolution of the undefined specifications

A.2.1. File name – file variable associations

Each Pascal file variable is associated with a named UNIX file. Except for *input* and *output*, which are exceptions to some of the rules, a name can become associated with a file in any of three ways:

1. If a global Pascal file variable appears in the **program** statement then it is associated with UNIX file of the same name.

2. If a file was reset or rewritten using the extended two-argument form of *reset* or *rewrite* then the given name is associated.
3. If a file which has never had UNIX name associated is reset or rewritten without specifying a name via the second argument, then a temporary name of the form 'tmp.x' is associated with the file. Temporary names start with 'tmp.1' and continue by incrementing the last character in the USASCII ordering. Temporary files are removed automatically when their scope is exited.

A.2.2. The program statement

The syntax of the **program** statement is:

program <id> (<file id> { , <file id > }) ;

The file identifiers (other than *input* and *output*) must be declared as variables of **file** type in the global declaration part.

A.2.3. The files input and output

The formal parameters *input* and *output* are associated with the UNIX standard input and output and have a somewhat special status. The following rules must be noted:

1. The program heading **must** contains the formal parameter *output*. If *input* is used, explicitly or implicitly, then it must also be declared here.
2. Unlike all other files, the Pascal files *input* and *output* must not be defined in a declaration, as their declaration is automatically:

   var input, output: text

3. The procedure *reset* may be used on *input*. If no UNIX file name has ever been associated with *input*, and no file name is given, then an attempt will be made to 'rewind' *input*. If this fails, a run time error will occur. *Rewrite* calls to output act as for any other file, except that *output* initially has no associated file. This means that a simple

   rewrite(output)

   associates a temporary name with *output*.

A.2.4. Details for files

If a file other than *input* is to be read, then reading must be initiated by a call to the procedure *reset* which causes the Pascal system to attempt to open the associated UNIX file for reading. If this fails, then a runtime error occurs. Writing of a file other than *output* must be initiated by a *rewrite* call, which causes the Pascal system to create the associated UNIX file and to then open the file for writing only.

A.2.5. Buffering

The buffering for *output* is determined by the value of the **b** option at the end of the **program** statement. If it has its default value 1, then *output* is buffered in blocks of up to 512 characters, flushed whenever a writeln occurs and at each reference to the file *input*. If it has the value 0, *output* is unbuffered. Any value of 2 or more gives block buffering without line or *input* reference flushing. All other output files are always buffered in blocks of 512 characters. All output buffers are flushed when the files are closed at scope exit, whenever the procedure *message* is called, and can be flushed using the built-in procedure *flush*.

An important point for an interactive implementation is the definition of 'input↑'. If *input* is a teletype, and the Pascal system reads a character at the beginning of execution to define 'input↑', then no prompt could be printed by the program before the user is required to type some input. For this reason, 'input↑' is not defined by the system until its definition is needed, reading from a file occurring only when necessary.

A.2.6. The character set

Seven bit USASCII is the character set used on UNIX. The standard Pascal symbols 'and', 'or', 'not', '<=', '>=', '<>', and the uparrow '↑' (for pointer qualification) are recognized.† Less portable are the synonyms tilde '~' for **not**, '&' for **and**, and '|' for **or**.

Upper and lower case are considered to be distinct. Keywords and built-in **procedure** and **function** names are composed of all lower case letters. Thus the identifiers GOTO and GOto are distinct both from each other and from the keyword **goto**. The standard type 'boolean' is also available as 'Boolean'.

Character strings and constants may be delimited by the character '' or by the character '#'; the latter is sometimes convenient when programs are to be transported. Note that the '#' character has special meaning when it is the first character on a line – see *Multi-file programs* below.

A.2.7. The standard types

The standard type *integer* is conceptually defined as

```
type integer = minint .. maxint;
```

Integer is implemented with 32 bit twos complement arithmetic. Predefined constants of type *integer* are:

```
const maxint = 2147483647; minint = -2147483648;
```

The standard type *char* is conceptually defined as

```
type char = minchar .. maxchar;
```

Built-in character constants are 'minchar' and 'maxchar', 'bell' and 'tab'; ord(minchar) = 0,

†On many terminals and printers, the up arrow is represented as a circumflex '^'. These are not distinct characters, but rather different graphic representations of the same internal codes. The proposed standard for Pascal considers them to be the same.

ord(maxchar) = 127.

The type *real* is implemented using 64 bit floating point arithmetic. The floating point arithmetic is done in 'rounded' mode, and provides approximately 15 digits of precision with numbers as small as 10 to the negative 308th power and as large as 10 to the 308th power.

A.2.8. Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)'. If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning will be printed if the sequence '(*' appears in a comment delimited by '(*' and '*)'. The restriction implied by this warning is not part of standard Pascal, but detects many otherwise subtle errors.

A.2.9. Option control

Options of the translators may be controlled in two distinct ways. A number of options may appear on the command line invoking the translator. These options are given as one or more strings of letters preceded by the character '–' and cause the default setting of each given option to be changed. This method of communication of options is expected to predominate for UNIX. Thus the command

```
tutorial% pi -l -s foo.p
```

translates the file foo.p with the listing option enabled (as it normally is off), and with only standard Pascal features available.

If more control over the portions of the program where options are enabled is required, then option control in comments can and should be used. One places the character '$' as the first character of the comment and follows it by a comma separated list of directives. Thus an equivalent to the command line example given above would be:

```
{$l+ ,s+ listing on, standard Pascal}
```

as the first line of the program. The 'l' option is more appropriately specified on the command line, since it is extremely unlikely in an interactive environment that one wants a listing of the program each time it is translated.

Directives consist of a letter designating the option, followed either by a '+ ' to turn the option on, or by a '–' to turn the option off. The b option takes a single digit instead of a '+ ' or '–'.

A.2.10. Notes on the listings

The first page of a listing includes a banner line indicating the version and date of generation of *pi* or *pc*. It also includes the UNIX path name supplied for the source file and the date of last modification of that file.

Within the body of the listing, lines are numbered consecutively and correspond to the line numbers for the editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a form-feed character, control-l, which causes a page eject in the listing, and a line with no characters which causes the line number to be suppressed in the listing, creating a truly blank line. These lines thus correspond to 'eject' and 'space' macros found in many

assemblers. Non-printing characters are printed as the character '?' in the listing.†

A.2.11. The standard procedure write

If no minimum field length parameter is specified for a *write*, the following default values are assumed:

integer	10
real	22
Boolean	length of 'true' or 'false'
char	1
string	length of the string
oct	11
hex	8

The end of each line in a text file should be explicitly indicated by 'writeln(f)', where 'writeln(output)' may be written simply as 'writeln'. For UNIX, the built-in function 'page(f)' puts a single ASCII form-feed character on the output file. For programs which are to be transported the filter *pcc* can be used to interpret carriage control, as UNIX does not normally do so.

A.3. Restrictions and limitations

A.3.1. Files

Files cannot be members of files or members of dynamically allocated structures.

A.3.2. Arrays, sets and strings

The calculations involving array subscripts and set elements are done with 16 bit arithmetic. This restricts the types over which arrays and sets may be defined. The lower bound of such a range must be greater than or equal to –32768, and the upper bound less than 32768. In particular, strings may have any length from 1 to 65535 characters, and sets may contain no more than 65535 elements.

A.3.3. Line and symbol length

There is no intrinsic limit on the length of identifiers. Identifiers are considered to be distinct if they differ in any single position over their entire length. There is a limit, however, on the maximum input line length. This limit is quite generous however, currently exceeding 160 characters.

†The character generated by a control-i indents to the next 'tab stop'. Tab stops are set every 8 columns in UNIX. Tabs thus provide a quick way of indenting in the program.

A.3.4. Procedure and function nesting and program size

At most 20 levels of **procedure** and **function** nesting are allowed. There is no fundamental, translator defined limit on the size of the program which can be translated. The ultimate limit is supplied by the address space. If one runs up against the 'ran out of memory' diagnostic the program may yet translate if smaller procedures are used, as a lot of space is freed by the translator at the completion of each **procedure** or **function** in the current implementation.

On the VAX-11, there is an implementation defined limit of 65536 bytes per variable. There is no limit on the number of variables.

A.3.5. Overflow

There is currently no checking for overflow on arithmetic operations at run-time on the PDP-11 or Sun. Overflow checking is performed on the VAX-11 by the hardware.

A.4. Added types, operators, procedures and functions

A.4.1. Additional predefined types

The type *alfa* is predefined as:

```
type alfa = packed array [ 1..10 ] of char
```

The type *intset* is predefined as:

```
type intset = set of 0..127
```

In most cases the context of an expression involving a constant set allows the translator to determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where it is not possible to determine the type of the set from local context, the expression type defaults to a set over the entire base type unless the base type is integer†. In the latter case the type defaults to the current binding of *intset*, which must be "type set of (a subrange of) integer" at that point.

Note that if *intset* is redefined via:

```
type intset = set of 0..58;
```

then the default integer set is the implicit *intset*.

A.4.2. Additional predefined operators

The relationals '<' and '>' of proper set inclusion are available. With *a* and *b* sets, note that

```
(not (a < b)) <> (a >= b)
```

†The current translator makes a special case of the construct 'if ... in [...]' and enforces only the more lax restriction on 16 bit arithmetic given above in this case.

As an example consider the sets *a* = [0,2] and *b* = [1]. The only relation true between these sets is '<>'.

A.4.3. Non-standard procedures

argv(i,a)	where *i* is an integer and *a* is a string variable assigns the (possibly truncated or blank padded) *i*'th argument of the invocation of the current UNIX process to the variable *a*. The range of valid *i* is *0* to *argc-1*.
date(a)	assigns the current date to the alfa variable *a* in the format 'dd mmm yy ', where 'mmm' is the first three characters of the month, i.e. 'Apr'.
flush(f)	writes the output buffered for Pascal file *f* into the associated UNIX file.
halt	terminates the execution of the program with a control flow backtrace.
linelimit(f,x)‡	with *f* a textfile and *x* an integer expression causes the program to be abnormally terminated if more than *x* lines are written on file *f*. If *x* is less than 0 then no limit is imposed.
message(x,...)	causes the parameters, which have the format of those to the built-in **procedure** *write*, to be written unbuffered on the diagnostic unit 2, almost always the user's terminal.
null	a procedure of no arguments which does absolutely nothing. It is useful as a place holder, and is generated by *pxp* in place of the invisible empty statement.
remove(a)	where *a* is a string causes the UNIX file whose name is *a*, with trailing blanks eliminated, to be removed.
reset(f,a)	where *a* is a string causes the file whose name is *a* (with blanks trimmed) to be associated with *f* in addition to the normal function of *reset*.
rewrite(f,a)	is analogous to 'reset' above.
stlimit(i)	where *i* is an integer sets the statement limit to be *i* statements. Specifying the **p** option to *pc* disables statement limit counting.
time(a)	causes the current time in the form ' hh:mm:ss ' to be assigned to the alfa variable *a*.

A.4.4. Non-standard functions

argc	returns the count of arguments when the Pascal program was invoked. *Argc* is always at least 1.
card(x)	returns the cardinality of the set *x*, i.e. the number of elements contained in the set.
clock	returns an integer which is the number of central processor milliseconds of user time used by this process.
expo(x)	yields the integer valued exponent of the floating-point representation of *x*; expo(*x*) = entier(log2(abs(*x*))).

‡Currently ignored by pdp-11 *px*.

random(x)	where *x* is a real parameter, evaluated but otherwise ignored, invokes a linear congruential random number generator. Successive seeds are generated as (seed*a + c) mod m and the new random number is a normalization of the seed to the range 0.0 to 1.0; a is 62605, c is 113218009, and m is 536870912. The initial seed is 7774755.
seed(i)	where *i* is an integer sets the random number generator seed to *i* and returns the previous seed. Thus seed(seed(i)) has no effect except to yield value *i*.
sysclock	an integer function of no arguments returns the number of central processor milliseconds of system time used by this process.
undefined(x)	a Boolean function. Its argument is a real number and it always returns false.
wallclock	an integer function of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970.