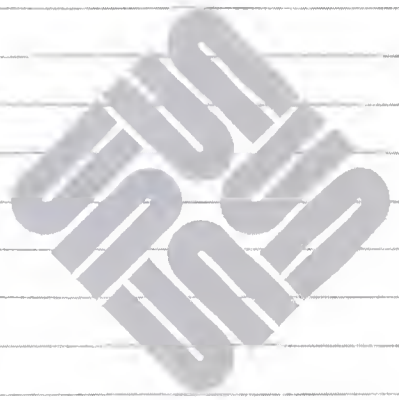
sun microsystems

Programming Tools *for the* Sun Workstation



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1180-01
Revision B of 15 May, 1985

Credits and Acknowledgements

The chapters of this manual were originally derived from the work of many people at Bell Laboratories, the 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.

Trademarks

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.
E	15 May 1985	Many minor corrections and updates. Extracted Assembly Language Reference Manual to make a separate manual.

Contents

Contents

Tables

Figures

Chapter 1

Programming the Shells

You can put a sequence of UNIX† commands in a file, and then you can get one of the Shells to read and execute the commands from the file. Such a file of UNIX commands is called a ***Shell script***.

Understand that Shell scripts do not serve the same function as the ***make*** program. ***Make*** 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 recreate these 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.

† UNIX is a trademark of Bell Laboratories.

Part I — Programming the C Shell

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

1.1. C-Shell 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* is automatically 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.

1.2. Variable Substitution in the C Shell

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 to generate 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 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.

1.3. Expressions in the C Shell

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.

1.4. Sample C 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*** with the named variable taking on each of the values given between '(' and ')' with the named variable — in this case 'i' is 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 it had during 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.[1]

---

[1] The Shell does not accept the following two formats:

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

and

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

```
if ( expression ) command
```

which can be written

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

The newline is escaped here for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command, and the final '\' must immediately precede the end-of-line.

The more general *if* statements also admits 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 the 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'. the ':h' modifier takes off the last component of a pathname leaving the head, and ':t' takes off all but the last component of a pathname leaving the tail. See the *csh* manual page in the *Commands Reference Manual for the Sun Workstation* for a full description of these modifiers.

It is also possible to use the *command substitution* mechanism to perform modifications on strings to then reenter 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[2].

Finally, note that the character '#' lexically introduces a Shell comment in a Shell script, 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.

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

[2] 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.

1.5. Other C-Shell 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 ***csh*** manual page 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 as they do in C, that is:

```
loop:
        commands
        goto loop
```

1.6. Supplying Input to Commands in the C Shell

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

Thus, a metanotation is used to supply in-line 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 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 substituting variables on the intervening lines. In general, the Shell uses << to terminate the text to be given to the command. If any part of the phrase following the << is quoted, these substitutions are not performed. In this case, since the form `1,$` was used in the editor script, you needed to ensure that the `$` is not variable-substituted. You can also ensure 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.

1.7. Catching Interrupts with 'onintr' in the C Shell

If your Shell script creates temporary files, you may wish to catch interrupts that occur while the Shell script is executing, so that 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 (status)
```

where *status* is the status you want to exit with.

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.

1.8. Other C-Shell Features

This section describes less commonly used C-Shell features.

1.8.1. Loops at the Terminal and Variables as Vectors

The *foreach* control structure aids in performing a number of similar commands. For instance, To count the number of persons using each Shell, you can say:

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

You can also use *foreach* to do this:

```
tutorial% foreach i ('csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
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 set the variable *a* to a list of all the filenames in the current directory. 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.

1.8.2. Command Substitution in the C Shell

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus, to save the name of 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 as arguments those files whose names end in '.c' which have the string 'TRACE' in them[3].

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 pages on *csh* and *sh* for details.

1.9. Special Characters in the C Shell

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

Syntactic Metacharacters	*Description*
;	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	*Description*
/	separates components of a file's pathname
.	separates root parts of a filename from extensions
?	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	*Description*
\	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	*Description*
<	indicates redirection of standard input
>	indicates redirection of standard output
>&	indicates redirection of standard output and standard error

[3] Command expansion also occurs in input redirected with '<<' and within '"' quotations. Refer to the *csh* manual page for full details.

Expansion/substitution Metacharacters	*Description*
$	indicates variable substitution
!	indicates history substitution
:	precedes substitution modifiers
^	used in special forms of history substitution
`	indicates command substitution

Other Metacharacters	*Description*
#	begins scratchfile names; indicates Shell comments
–	prefixes option (flag) arguments to commands
%	prefixes job name specifications

Part II — Programming the Bourne Shell

1.10. Bourne 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 (***cheese*** say) to the null string, you can say:

```
cheese=
```

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

```
$ echo $user
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
```

redirects it to the file whose name is ***tmpa***.

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

Variable	*Explanation*
`$?`	The exit status (return code) of the last command executed, as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of a return code is dealt with later under `if` and `while` commands.
`$#`	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 the Shell is used interactively, it looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the Shell prints the message ***you have mail*** before prompting for the next command. This variable is typically set in the file *.profile,* in the user's login directory. For example:

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

`$HOME` The default argument for the *cd* command. The command *cd* with no argument is equivalent to:

```
cd $HOME
```

This variable is also typically set in *.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. `$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.

`$PS1` The primary Shell prompt string, by default, '`$ `'.

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

`$IFS` The set of characters used by ***blank interpretation.***

1.11. Control Flow in the Bourne Shell — 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* which searches the file */usr/lib/telnos* which contains lines of the form

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

The text of *tel* is

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

The command

```
$ tel fred
```

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

```
$ 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:

```
$ 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`.

1.12. Control Flow in the Bourne Shell — case

The `case` notation provides a multi-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

```
$ 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:

```
$ append file1 file2
$
```

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 ?.

1.13. Here Documents in the Bourne Shell

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 by 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.

1.14. The 'test' Command

Although the *test* command is not part of the Shell, Shell programs frequently use it. 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

1.15. Control Flow in the Bourne Shell — 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-list1
do command-list2
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 discards `$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.

1.16. Control Flow in the Bourne Shell — if

A general conditional branch of the form

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

is also available to test 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:

```
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.

1.17. 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*.

1.18. Debugging Bourne Shell Procedures

The Shell provides two tracing mechanisms to help in debugging Shell procedures. The first is invoked within a 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
```

produces an execution trace. Following parameter substitution, each command is displayed as it is executed. Both flags may be turned off by saying

```
set -
```

and the current setting of the Shell flags is available as `$-`.

1.19. The 'man' Command

The *man* command displays pages from the on-line manuals. *Man* 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.

Figure 1-1: A version of the man command

```
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
```

1.20. Keyword Parameters in the Bourne Shell

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 –.

1.21. Parameter Transmission in the Bourne Shell

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 exported 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 values are 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.

1.22. Parameter Substitution in the Bourne Shell

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

```
$ echo $d
```

or

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

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

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

which will echo the value of the variable **d** if it is set and '.' otherwise. The default string is

evaluated using the usual quoting conventions so that

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

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

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

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

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

which substitutes the same string as

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

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

If there is no sensible default then the notation

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

echos the value of the variable d if it has one; otherwise the Shell prints *message*, if the Shell if not interactive, and stops executing the procedure. 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, and the Shell is not interactive, the Shell stops executing the procedure.

1.23. Command Substitution in the Bourne Shell

In a similar way, you can substitute the standard output from a command as the value of a parameter. 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[4] (`...`) 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"`
```

[4] Often called backquotes.

is equivalent to

```
ls $1
```

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

```
basename /usr/fred/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 pathname and 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, for instance, *1977 Nov 1, 23:59:59*

1.24. Evaluation and Quoting in the Bourne Shell

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 filenames 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 parameter and command 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 special meanings within double quotes and may be quoted using \.

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

For example,

```
echo "$x"
```

passes the value of the variable `x` as a single argument to *echo*. Similarly,

```
echo "$*"
```

passes 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 "$@"
```

passes 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.

Table 1-1: Quoting Mechanisms

Quoting Character	*Metacharacter*					
\	$	*	`	"	'	
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

Where t=terminator, y=interpreted, and n=not interpreted

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 `y` has the value *pqr*, then

```
eval echo $X
```

echos 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) are 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.

1.25. Error Handling in the Bourne Shell

The treatment of errors detected by the Shell depends on the type of error and on whether the Shell is being used interactively. A Shell invoked with the `-i` flag is deemed to be 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 an 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.

Table 1-2: UNIX Signals

Signal Number	*Description*
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))

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.

1.26. Fault Handling in the Bourne Shell

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 it executes 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 its 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 3). The cleanup action is to remove the file **junk$$**.

Figure 1-2: The touch Command

```
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
```

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 4) 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.

Figure 1-3: The scan Command

```
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
```

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

1.27. Command Execution in the Bourne Shell

To run a command (other than a built-in), the Shell first creates a new process using the *fork* system call. 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 specified signals 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 quoted with backslashes are 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. First, the default standard input for such a command is the empty file ***/dev/null***. This prevents two processes (the Shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
$ ed file &
```

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

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that 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.

1.28. Calling the Bourne 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, 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 determined 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.

1.29. Bourne Shell 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

1.30. Bourne Shell 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
{  }
read
```

Chapter 2

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.

Section 2.8 — *The Standard I/O Library* — describes the standard I/O library in detail.

This chapter describes how to write programs that interface with the UNIX operating system in a nontrivial 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]. You should also be familiar with UNIX itself, at least to the level of *UNIX for Beginners* [3].

2.1. Basics

2.1.1. 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.

```
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 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 `argv[argc-1]`.

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.

2.2. The 'Standard Input' and 'Standard Output'

The simplest input mechanism is to read from 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 (input redirection): if `prog` uses `getchar`, the command line

```
tutorial% prog < filename
```

makes `prog` read from the file specified by ***filename*** 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 (see below) 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 > outputfile
```

writes the standard output on ***outputfile*** instead of the terminal. ***outputfile*** is created if it doesn't exist; if it already exists, its previous contents are overwritten. 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 which does I/O using the standard I/O functions described in section 3(S) of the ***System Interface Manual*** — the C compiler reads 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 2.5.3 discusses returning status in more detail.

2.3. 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.

This section discusses the basics of the standard I/O library. Section 2.8 — ***The Standard I/O Library*** — contains a more complete description of its capabilities and calling conventions.

2.3.1. Accessing Files

The above programs have all read the standard input and written the standard output, which we have assumed are magically predefined. 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
```

displays 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 filenames to the I/O statements which actually read the data.

The rules are simple — you have to ***open*** a file by the standard library function `fopen` before it can be read from or written to. `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 has the form:

```
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 allowable modes are read (`"r"`), write (`"w"`), or append (`"a"`). In addition, each ***mode*** may be followed by a + sign to open the file for reading and writing. `"r+"` positions the stream at the beginning of the file, `"w+"` creates or truncates the file, and `"a+"` positions the stream to the end of the file. Both reads and writes may be used on read/write streams, with the limitation that an `fseek`, `rewind`, or reading end-of-file must be used between a read and a write or vice versa.

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing discards the old contents. 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` returns the null pointer value `NULL` — 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` as its value. `getc` and `putc` return `EOF` on error.

When a program is started, three streams are opened automatically, and file pointers are provided for them. These streams 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 predefined 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 standalone 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 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.

2.3.2. 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, unless the standard error is also 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 argument of `exit` is made available to whatever process called the process that is exiting (see Section 2.5.3, so the success or failure of the program can be tested by another program that uses this one as a subprocess. By convention, a return value of 0 signals that all is well; nonzero 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` terminates the program immediately without any buffer flushing; it may be called directly if desired.

2.3.3. Miscellaneous I/O Functions

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

Normally output with `putc`, and such is buffered — use `fflush(fp)` to force it out immediately.

`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`, and no input or output is done.

`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.

2.4. Low-Level Input Output

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.

2.4.1. 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? — 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 2.3 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 standard input, standard output, and 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 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.

2.4.2. 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 1024, corresponding 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 1024

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, and 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 Sun 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 1024

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);
}
```

2.4.3. 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`.

`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 rewrite old ones.

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

returns a file descriptor if it could 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 1024
#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 20-32) 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 reuse 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.

2.4.4. 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));
}
```

2.4.5. 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 variable `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 display the reason for failure. The routine `perror` displays 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 displayed by your program.

2.5. 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.

2.5.1. 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 timestamp 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 that `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 section 2.8.

2.5.2. 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[5].

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

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

The first argument to `execl` is the ***filename*** 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 placeholder. 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 start the second pass simply by 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, you might try

```
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);
```

---

[5] `system` uses */bin/sh* (the Bourne Shell) to execute the command string, so syntax specific to the C-Shell will not work.

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`.

2.5.3. 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 nonzero; 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 nonzero 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 nonzero 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 to point at the right files (see Section 2.4.1), 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.

2.5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other process reads from the pipe. 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 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 waits until data arrives; if a process writes into a pipe which is too full, it waits 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 pipe with a `pipe` system call; it then `fork`'s 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 `close`'s 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[6]. Finally, the old read side of the pipe is closed.

[6] Yes, this is a bit tricky, but it's a standard idiom.

A similar sequence of operations takes place when the child process is supposed to write to 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 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.

2.6. 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*** and ***quit***, which are generated from the keyboard[7], ***hangup***, caused by hanging up the phone on dialup lines, and ***terminate***, generated by the ***kill*** command. When one of these events occurs, the

[7] The current binding of characters and signals can be discovered by the `stty all` command. On Sun systems, typing control-C usually generates the *kill* signal and control-\ generates the *quit* signal.

signal is sent to *all* processes which were started from the corresponding terminal — the signal terminates the process unless other arrangements have been made. In the *quit* case, a core image file is written for debugging purposes.

`signal` is the routine which alters the default action. `signal` has two arguments: the first specifies the signal to be processed, and the second argument specifies what to do with that signal. The first argument is just a numeric code, but the second 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);
```

means that interrupts are 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 so that the program can 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 display should not terminate the edit session 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. The `setjmp` routine then saves the state of things. When an interrupt occurs the `onintr` routine is called, which can display a message, set flags, or whatever. `longjmp` takes as argument an object set by `setjmp`, and restores control to the location following 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 when a signal occurs sets a flag and then returns instead of calling `exit` or `longjmp`, execution continues 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** routine 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 catching signals is combined with executing 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 Sun system — 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
```

2.7. References

[1] Sun Microsystems Reference Manuals: *Commands Reference Manual* and *System Interface 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.

2.8. 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 non-Sun running a version of UNIX.

2.8.1. 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 stream

`stdout` the name of the standard output stream

`stderr` the name of the standard error stream

`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. 1024) 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 constants and may not be assigned to.

2.8.2. Standard I/O Library Calls

```
FILE *fopen(filename, type)
    char *filename;
    char *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. In addition, each ***mode*** may be followed by a + sign to open the file for reading and writing. `"r+"` positions the stream at the beginning of the file, `"w+"` creates or truncates the file, and `"a+"` positions the tream to the end of the file. Both reads and writes may be used on read/write streams, with the limitation that an `fseek`, `rewind`, or reading end-of-file must be used between a read and a write or vice versa. The value returned is a file pointer. If it is NULL the attempt to open failed.

```
FILE *freopen(filename, type, ioptr)
    char *filename;
    char *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` is returned, which now refers to the new file. Often the reopened stream is `stdin` or `stdout`. The `filename` and `type` parameters are as for `fopen`.

```
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.

```
int putc(c, ioptr)
    int  c;
    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, and EOF is returned on error.

```
int fputc(c, ioptr)
    int  c;
    FILE *ioptr;
```

acts like `putc` but is a genuine function, not a macro.

```
int 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.

```
int fflush(ioptr)
    FILE *ioptr;
```

Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if they are not directed to the terminal.

```
(void) exit(errcode);
    int 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`.

```
int feof(ioptr)
    FILE *ioptr;
```

returns nonzero when end-of-file has occurred on the specified input stream.

```
int ferror(ioptr)
    FILE *ioptr;
```

returns nonzero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

```
int getchar();
```

is identical to `getc(stdin)`.

```
int putchar(c);
```

is identical to `putc(c, stdout)`.

```
char *fgets(s, n, ioptr)
    char *s;
    int  n;
    FILE *ioptr;
```

reads to `n-1` characters, or up to a newline character, whichever comes first, from the stream `ioptr` into the string pointed to by the character pointer `s`. A null character is placed after the last character read in the strings `s`. `fgets` returns the first argument, or NULL if error or end-of-file occurred.

```
int puts(s)
    char *s;
```

`puts` copies the null-terminated strings specified by `s` onto the standard output stream and appends a newline character.

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

writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. The last character transmitted is returned as value, or EOF is returned on error.

```
int ungetc(c, ioptr)
    int  c;
    FILE *ioptr;
```

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

```
int printf(format, a1, ...)
    char *format;

int fprintf(ioptr, format, a1, ...)
    FILE *ioptr;
    char *format;

int sprintf(s, format, a1, ...)
    char *s;
    char *format;
```

`printf` writes on the standard output. `fprintf` writes on the output stream named by `ioptr`. `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*.

All three functions return the number of characters actually transmitted, except that `printf` and `fprintf` return EOF if any error condition exists on the output file.

```
int scanf(format, a1, ...)
    char *format;

int fscanf(ioptr, format, a1, ...)
    FILE *ioptr;
    char *format;

int sscanf(s, format, a1, ...)
    char *s;
    char *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 the `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.

```
int fread(ptr, sizeof(*ptr), nitems, ioptr)
    unsigned nitems;
    FILE *ioptr;
```

reads `nitems` of data of the type of `*ptr` from file `ioptr` into the memory area starting at `ptr`. No advance notification that binary I/O is being done is required. `fread` returns the number of items actually read from the specified stream.

```
int fwrite(ptr, sizeof(*ptr), nitems, ioptr)
    unsigned nitems;
    FILE *ioptr;
```

Like `fread`, but in the other direction. `fwrite` returns the number of items actually transmitted to the specified stream. This may possibly be less than the number of items requested if an error occurs while the transfer is in process.

```
(void) 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.

```
int system(string)
    char *string;
```

The `string` is executed by the shell as if typed at the terminal. The return value is the exit code of the invoked shell, which is usually the exit code of the last command executed by it.

```
int 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 32 bits on the Sun Workstation.

```
int putw(w, ioptr)
    FILE *ioptr;
```

writes the integer `w` on the named output stream. `putw` returns the current error status of the specified stream, as if an `ferror` call had been made.

```
(void) 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 is unbuffered. Otherwise the buffer supplied is used. It must be a character array of sufficient size:

```
char    buf[BUFSIZ];

(void) setbuffer(ioptr, buf, size)
    FILE *ioptr;
    char *buf;
    int  size;
```

`setbuffer` is like `setbuf` (described above), but can be used when a specified, nonstandard buffer size should be used.

```
int fileno(ioptr)
    FILE *ioptr;
```

returns the integer file descriptor associated with the file.

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

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.

```
int getpw(uid, buf)
    int uid;
    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)
    int  num;
```

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

```
int free(ptr)
    char  *ptr;
```

`free` frees up memory previously allocated by `malloc`. `free` returns a 0 if any errors were detected (such as `ptr` being misaligned), and returns 1 otherwise. Disorder can be expected if the pointer was not obtained from `malloc`.

```
char *calloc(num, size);
    unsigned num;
    unsigned size;
```

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

```
(void) cfree(ptr, num, size)
    char *ptr;
    unsigned num;
    unsigned size;
```

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 nonzero if *c* is alphabetic.

`isupper(c)` returns nonzero if *c* is upper-case alphabetic.

`islower(c)` returns nonzero if *c* is lower-case alphabetic.

`isdigit(c)` returns nonzero if *c* is a digit.

`isxdigit(c)` returns nonzero if *c* is a hexadecimal digit — that is, one of '0' through '9', 'a' through 'f', or 'A' through 'F'.

`isspace(c)` returns nonzero if *c* is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns nonzero if *c* is any punctuation character, that is, not a space, letter, digit or control character.

`isalnum(c)` returns nonzero if *c* is a letter or a digit.

`isprint(c)` returns nonzero if *c* is printable — a letter, digit, space, or punctuation character.

`iscntrl(c)` returns nonzero if *c* is a control character.

`isascii(c)` returns nonzero if *c* is an ASCII character, that is, less than octal 0200.

`isgraph(c)` returns nonzero if *c* is a printing character — like `isprint(c)` but doesn't include the space character.

`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*.

Chapter 3

Lint — A C Program Checker

Lint examines C source programs, detecting a number of bugs and obscurities. *Lint* enforces the type rules of C more strictly than the C compiler. *Lint* 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 its implementation, and gives some hints on writing machine-independent C code.

3.1. Using Lint

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

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

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

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

produces, in addition to the types of messages described above, additional messages relating to portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** produces 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. There is a summary of *lint* options in section 3.21.

3.2. 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,' which is 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 '*xxx* 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.

3.3. Unused Variables and Functions

As 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 make working programs 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** option 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** option may be used to suppress the spurious messages which might otherwise appear.

3.4. 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.

3.5. Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It complains 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 that *lint* does not complain about is a `break` statement that cannot be reached — programs generated by *yacc*[2], and especially *lex*[3], may have literally hundreds of unreachable `break` statements. The **−O** option in the C compiler often eliminates 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.

3.6. Function Values

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

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

```
return( expr );
```

and:

```
return;
```

statements results in 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!

3.7. Type Checking

Lint enforces the type checking rules of C more strictly than the compiler does. 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.

3.8. Type Casts

The type casting 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` option 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.

3.9. Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from −128 to 127. In most other C implementations, characters take on only positive values. Thus, *lint* will mark 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`.

3.10. Assignments of longs to ints

Bugs may arise from the assignment of a `long` to an `int`, which may lose 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 `int`'s, 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** option.

3.11. Strange Constructions

Lint flags several perfectly legal, but somewhat strange, constructions — it is hoped that the messages encourage better code quality, clearer style, and may even point out bugs. The **−h** option 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 reports '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** option 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.

3.12. 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 result in 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., and the Sun C compiler issues warning messages about them.

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

```
int  x  1 ;
```

to initialize *x* to 1, also creating 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.

3.13. 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** options are in effect.

3.14. 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
```

3.15. 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 many C compilers, including Sun's. 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 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*.

3.16. 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 resolves these declarations, and sets aside only a single word of storage for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration sets aside a word of storage called *a*. When loading or library editing takes place, this creates fatal conflicts which prevent the proper operation of the program. *Lint* detects such multiple definitions if it is invoked with the **−p** option.

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** maps all external symbols to one case and truncates them 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 option multi-character character constants.

Of course, the word sizes are different! This is less troublesome 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.

3.17. 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 make *lint* recognize a number of words 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 placing the directive

```
/* NOTREACHED */
```

just before that 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** option can be turned on for one function by the directive:

```
/* ARGSUSED */
```

Complaints about variable numbers 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 */
```

checks the first two arguments and leaves 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.

3.18. 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 in a library file, but not used in 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 routines 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** option 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** option can be used to suppress all library checking.

3.19. 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 **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.

3.20. 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).

3.21. Current Lint Options

The *lint* command currently has the form

```
tutorial% lint [-abchnpsuvx ] filename... library-descriptors...
```

The options are

a Report assignments of `long` to `int` or shorter

b Report unreachable `break` statements

c Complain about questionable casts

h Perform heuristic checks

n Do not do library checking

p Perform portability checks

s Same as **h** (for historical reasons)

u Don't report unused or undefined externals

v Don't report unused arguments

x Report unused external declarations

Chapter 4

Make — A Program for Maintaining Computer Programs

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, and some may need to be processed by a sophisticated program generator (for example, Yacc[1] or Lex[2]). The outputs of these generators may 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 usually results 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.

Make mechanizes many of the activities of program development and maintenance. *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.

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[8] or of describing huge programs.

4.1. 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 ***lm*** library. By convention, output of the C compilations is 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   -lm  -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), filenames 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:

[8] See the description of the Source Code Control System (SCCS) later in this book, for a tool for maintaining multiple source versions.

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -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. $$ produces 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 are discussed below. The following fragment shows how macros are used:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)   $(LIBES)   -o prog
  . . .
```

The command:

```
tutorial% make
```

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

```
tutorial% make  "LIBES= -ll -lm"
```

loads them with both the *lex* (-ll) and mathematical (-lm) 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.

4.2. 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 (#) to the end of the line 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, but trailing ones are not). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
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 displayed and then passed to a separate invocation of the Shell after substituting for macros. The displaying 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.

Make ignores errors if the **–i** option 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 — these criteria are necessary because 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.

Make sets certain macros before issuing any command. $@ 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 filenames.

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*** displays a message and stops.

4.3. Command Usage

The ***make*** command takes three kinds of arguments: macro definitions, options, and target filenames.

tutorial% make [*options*] [*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 options are examined. The permissible options are:

-i Ignore error codes returned by invoked commands. This mode is also entered if the fake target name .IGNORE appears in the description file.

-s Do not display 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 Display commands, but do not execute them. Even lines beginning with an '@' sign are displayed.

-t Touch the target files (bringing them to be up-to-date) rather than issuing 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 Display the complete set of macro definitions and target descriptions.

-d Debug mode. Display detailed information on files and times examined.

-f *filename*
Use *filename* as the name of the description file. A file name of - denotes the standard input. In the absence of the **-f** option, *make* first looks for a file called *makefile* in the current directory, then looks for a file called *Makefile* in the current directory. The contents of the description files override the built-in rules if they are present.

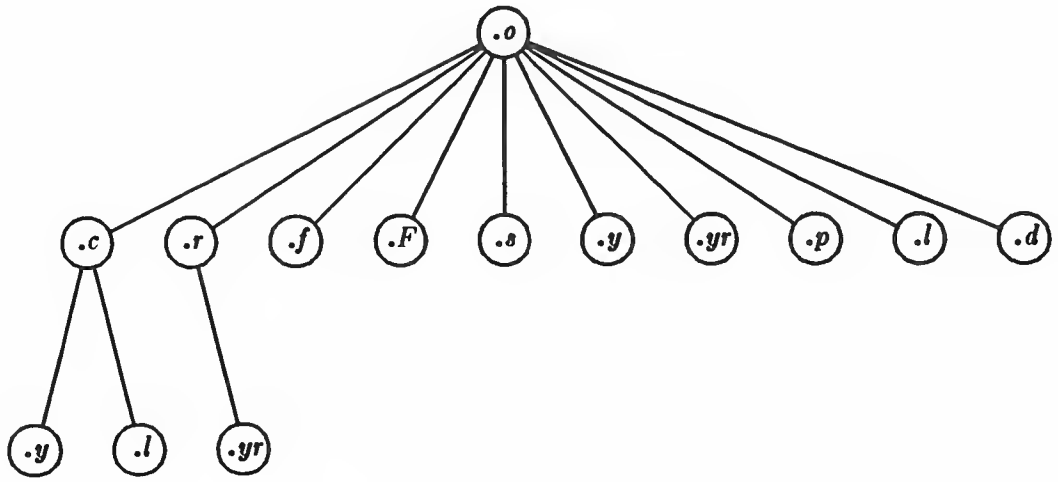
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left-to-right order. If there are no such arguments, the first name in the description files that does not begin with a period is 'made'.

4.4. 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. Section 4.7 describes these tables and means of overriding them. The default suffix list is:

Suffix	*Type of File*
.o	Object file
.c	C source file
.r	Ratfor source file
.f	Fortran source file
.F	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.p	Pascal source
.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 option arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, FC, PC, RC, YACC, YACCR, and LEX. The command:

```
tutorial% make CC=newcc
```

uses the *newcc* command instead of the usual C compiler. The macros CFLAGS, FFLAGS, PFLAGS, RFLAGS, YFLAGS, and LFLAGS may be set to issue these commands with optional options.

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

uses the optimizing C compiler.

The *make* variable MFLAGS is also useful — it contains a list of the command-line arguments given to this invocation of *make*.

4.5. Example

As an example of the use of *make*, consider the following description file which could be 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
LINT = lint -p
CFLAGS = -O
make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) -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 displays 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 -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 @ sign on the *size* command in the description file suppressed the displaying of the *size* command, so only the sizes are displayed.

The last few entries in the description file are useful maintenance sequences. The 'print' entry displays 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 last touched. The 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"
```

4.6. Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a `#include "defs"` line, 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 display the commands it would issue without actually executing 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') makes the relevant files 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 option (**−d**) generates a very detailed description of what *make* is doing, including the file times. The output is verbose, and recommended only as a last resort.

4.7. Suffixes and Transformation Rules

Make itself does not know what filename 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** option 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 are 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:

```
Macros:
        LOADLIBES =
        FFLAGS =
        FC = f77
        RFLAGS =
        RC = f77
        CFLAGS =
        PFLAGS =
        PC = pc
        AS = as
        CC = cc
        LFLAGS =
        LEX = lex
        YFLAGS =
        YACCR = yacc -r
        YACC = yacc
        MFLAGS = -p
        $ = $

.y.c:                                   Rules for converting yacc grammar to C code
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@

.l.o:                                   Rules for converting Lex grammar to object code
        $(LEX) $(LFLAGS) $<
        $(CC) $(CFLAGS) -c lex.yy.c
        rm lex.yy.c
        mv lex.yy.o $@

.yr.o:                                  Rules for converting Ratfor yacc grammar to object code
```

```
        $(YACCR) $(YFLAGS) $<
        $(RC) $(RFLAGS) -c y.tab.r
        rm y.tab.r
        mv y.tab.o $@

.y.o:                                   Rules for converting yacc grammar to object code
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@

.s.o:                                   Rules for converting assembler source to object code
        $(AS) -o $@ $<

.f.o:                                   Rules for converting FORTRAN-77 source to object code
        $(FC) $(RFLAGS) $(FFLAGS) -c $<

.F.o:                                   Rules for converting FORTRAN-77 source to object code
        $(FC) $(RFLAGS) $(FFLAGS) -c $<

.r.o:                                   Rules for converting Ratfor source to object code
        $(FC) $(RFLAGS) $(FFLAGS) -c $<

.p.o:                                   Rules for converting Pascal source to object code
        $(PC) $(PFLAGS) -c $<

.c.o:                                   Rules for converting C source to object code
        $(CC) $(CFLAGS) -c $<
```

4.8. Acknowledgments and References

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*.

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.

Chapter 5

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 text file; 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 alternative 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 which 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

The 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 rather than how SCCS works; for this reason some of the examples are not well explained. For details of what the magic options do, see Part II of this manual.

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

5.1. Learning the Lingo

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

5.1.1. S-file

The ***s-file*** is a single file that holds all the different versions of your file. The ***s-file*** contains only the the original version and 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.

5.1.2. 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[9]. 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 later changes.

5.1.3. SID's (version numbers)

An 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.

5.1.4. 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 in-line 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% produces 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% results in the date the latest delta was 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).

---

[9] 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.

5.2. 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 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-file*s, 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.

5.3. 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 has been retrieved[10] 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*.

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

5.4. Changing Files (Creating Deltas)

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

To edit a source file, you must first get it, requesting permission to edit it[11]. 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 text editor:

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

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

When the desired changes have been 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* then types:

```
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[12]. The *prog.c* file is then removed; it can be retrieved using *get*.

5.4.3. 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.

---

[11] The *edit* command is equivalent to using the –e option to *get*, as:

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

Keep this in mind when reading other documentation.

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

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

To find out what files are being edited, type:

```
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 delta'ed.

If you know that everything being edited should be delta'ed, 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** option to ignore 'branches' (alternate versions, described later) and the **−u** option to give only files being edited by you. The **−u** option 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.

5.4.5. 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.

5.4.5.1. Finding Out What Versions Are Being Used 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 one can see that the source in ***prog.c*** will not compile into the same version as the

binary in */usr/bin/prog*.

5.4.5.2. 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 they let you find out what version of the object is being run. However, there is a cost: data space is used up to store the keywords.

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.

5.4.6. 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 redelta'ed. 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.

5.4.7. 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
```

5.5. Restoring Old Versions

5.5.1. 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) option. For example,

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

retrieves 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 as:

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

5.5.2. 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 through 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 displays 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.

5.6. Auditing Changes

5.6.1. Displaying Delta Comments with 'sccs prt'

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

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

which produces 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
```

5.6.2. 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 changes were made by this delta 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** option makes SCCS output the generated source to the standard output rather than to a file.

5.6.3. 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" options can be used. To pass the **-c** option, 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.

5.7. Shorthand Notations

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

5.7.1. 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:

```
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.

5.7.2. 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** option 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.

5.7.3. Backing Out of 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
```

5.7.4. Working From Other Directories with the -d Flag

If you are working on a project where the SCCS code is in a directory somewhere else, 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** option.

5.8. 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 delta'ed before installing the program for general use. This will ensure that it is possible to restore any version in use at any time.

5.9. Saving Yourself

5.9.1. Recovering a Munged Edit File

Sometimes you may find that you have destroyed or trashed a file that you were trying to edit[13]. 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.

5.9.2. 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
```

5.10. 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:

[13] 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** option 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.

-t*file*
store descriptive text from ***file*** in the SCCS file. This descriptive text might be the documentation or a design and implementation document. Using the **-t** option ensures that if the SCCS file is passed on to someone else, the documentation will go along with it. 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.

5.11. 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** option can be specified when the SCCS file is first created.

5.11.1. 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*.

5.11.2. 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
```

5.11.3. 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 released 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.

5.11.4. 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** option 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** option to ***edit*** as described above.

5.11.5. 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.

5.12. 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 below.

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 their 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 unwanted files 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 removes all files for which an *s-file* exists, but which are not being edited.

5.12.1. 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.

5.12.2. 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 separate from 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-file*'s are up-to-date (that is, nothing is being edited), and will abort the *make* if this condition is not met.

5.12.3. 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 *Makefile*s 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 the above, the *touch* command can be removed completely; the equivalent effect will be achieved by doing an automatic *get* on *z.h*.

5.13. Quick Reference

5.13.1. Commands

The following commands should all be preceded with 'sccs'. This list is not exhaustive; for more options see Part II of this manual.

`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 for ***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 for ***get***.
- **-x***list* Same as for ***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-file*'s.

−i*file*	Create, using *file* as the initial contents.
−z	Rebuild the checksum in case the file has been trashed.
−f*flag*	Turn on *flag*.
−d*flag*	Turn off (delete) *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 and implementation documents to ensure they get distributed with the *s-file*.
	Useful flags that can be introduced via the **−F** and **−d** options are:
b	Allow branches to be made using the **−b** option to *edit*.
d*SID*	Default SID to be used on a *get* or *edit*.
i	Make the 'No Id Keywords' error message 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*.

5.13.2. 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.

5.14. 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 **`bold typewriter font like this`**, and what the terminal displays shown in `typewriter font like this`. After familiarizing yourself with basics, use the manual pages for detailed SCCS command descriptions.

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.

5.14.1. 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.

5.14.2. 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 easily be reconstructed with the ***get*** command, described below.

5.14.3. 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%
```

5.14.4. 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'.

5.14.5. 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.

5.14.6. 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.

5.15. 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.

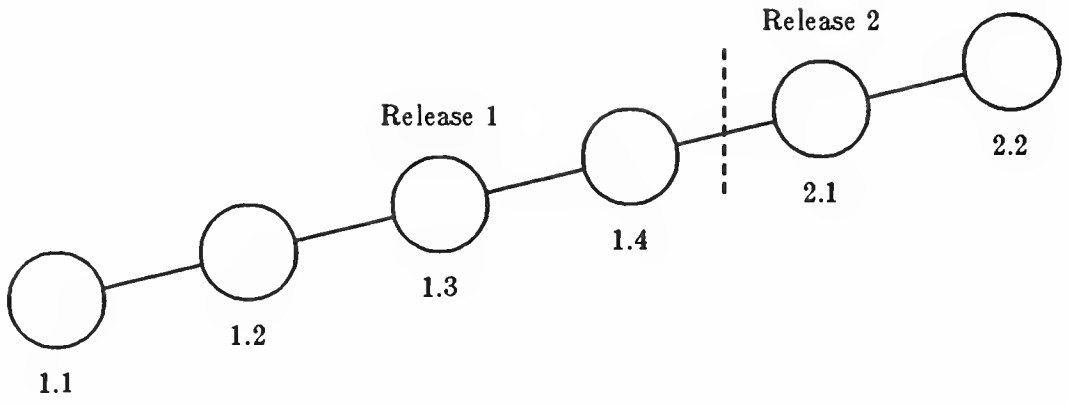


Figure 5-1: Evolution of an SCCS File

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*** is needed on the tree: 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.

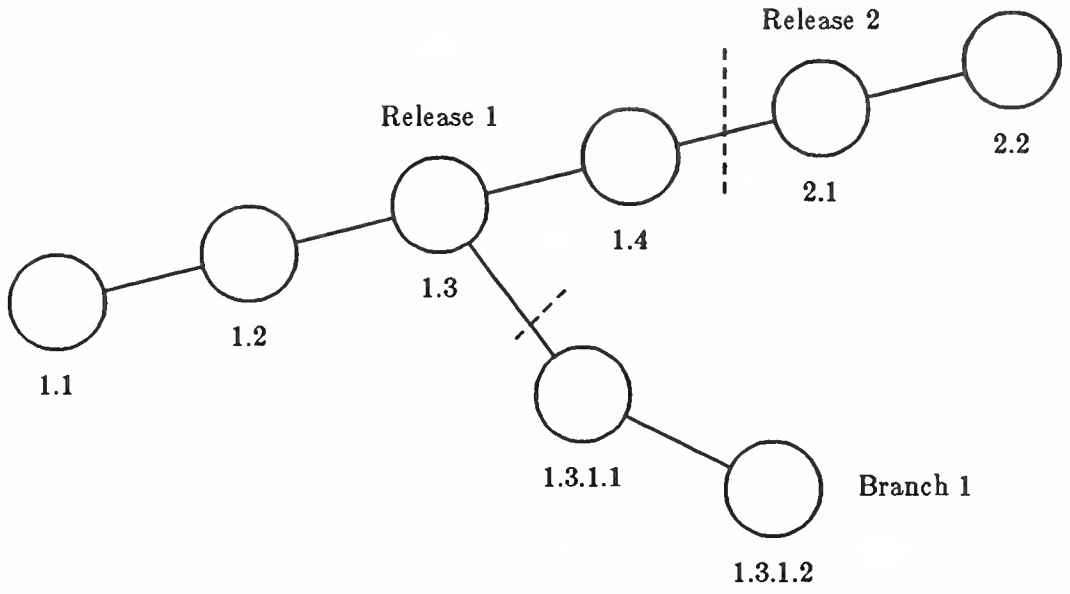


Figure 5-2: Tree Structure with Branch Deltas

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 a branch delta 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).

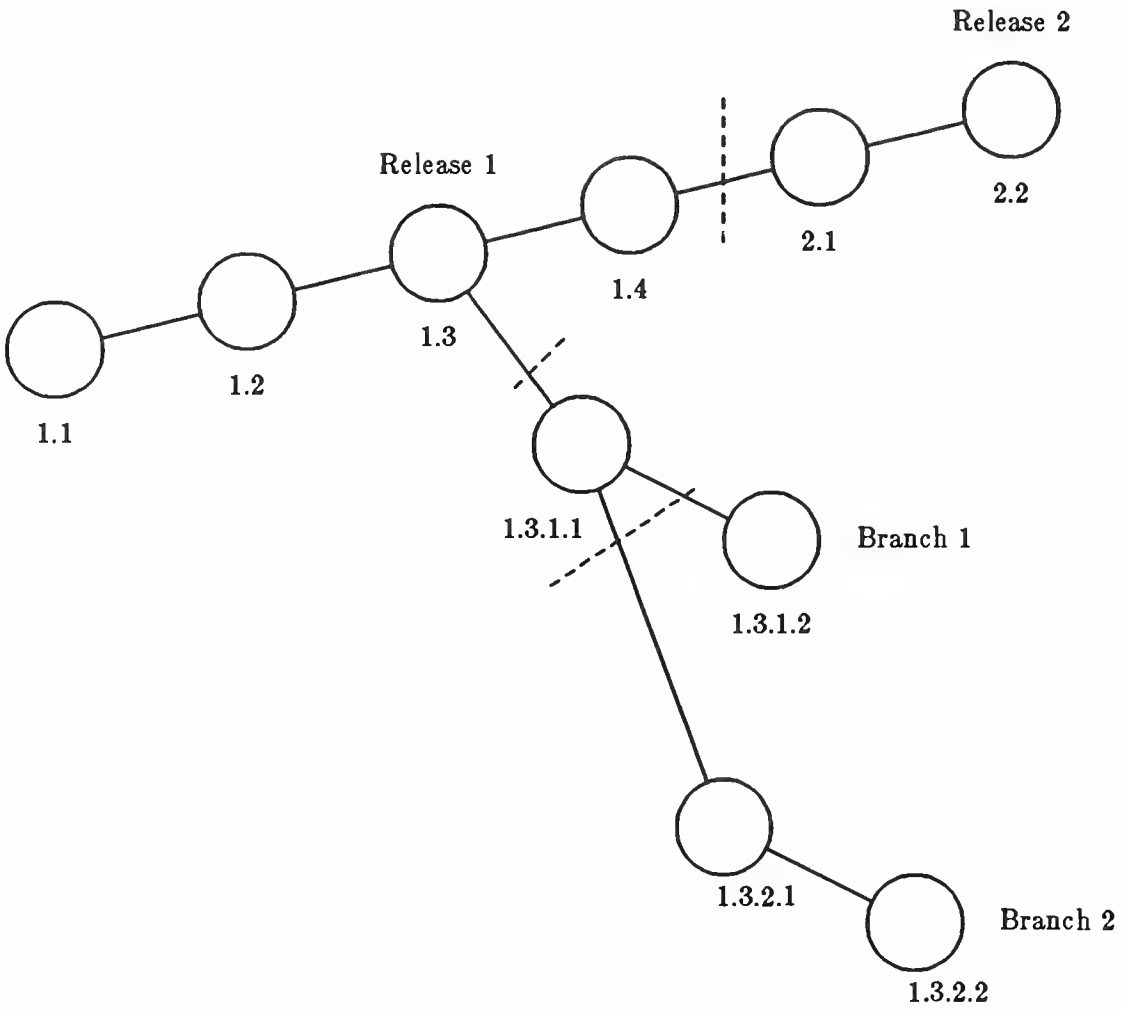


Figure 5-3: Extending the Branching Concept

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.

5.16. 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.

5.16.1. 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[14] 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.

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

5.16.2. 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).

5.16.3. 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 the system.

5.16.4. 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

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

system crashes or similar situations.

5.16.5. 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.

5.17. SCCS Commands

This section describes the major features of all the SCCS commands. For detailed descriptions of the commands and 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.

5.17.1. 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%
```

5.17.1.1. 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 an ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is an ID keyword for the current date (in the form 'mm/dd/yy'), and %M% is 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).

5.17.1.2. 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 an 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%
```

5.17.1.3. 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:

   `floor` $\leq$ `R` $\leq$ `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 fails.

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.

5.17.1.4. 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 this 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':[15]

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get*.

The first execution of `get -e` *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 overwrite 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[16] 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*.

[15] Other information may be present, but is not of concern here. See *get*(1) for further discussion.

[16] 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.

5.17.1.5. 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.

5.17.1.6. 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 file 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 has been 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** option may be used with the **-e** option.

See *get*(1) for a full description of additional *get* options.

5.17.2. 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[17].
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

[17] The SID specified may be either the SID retrieved by *get*, or the SID *delta* is to create.

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[18] numbers, separated 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 regardless (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***[19]. If there is only ***one*** entry in the ***p-file***, the ***p-file*** itself is removed.

---

[18] 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 delta.

[19] All updates to the ***p-file*** are made to a temporary copy, the ***q-file***, whose use is similar to the use of the ***z-file*** described above.

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.

5.17.3. 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.

5.17.3.1. 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 the previous 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 meaningful only in creating the first delta.

5.17.3.2. 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.

5.17.3.3. Initializing and Modifying SCCS File Parameters

The portion of the SCCS file reserved for ***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 is to 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 filename 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** (**f**lag) option, or may be deleted with the **−d** (**d**elete) 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** (**m**odule 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.

5.17.4. 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***[20] 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%
```

[20] Not to be confused with *get ID keywords*.

-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.

5.17.5. 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 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%
```

5.17.6. 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:

   ```
   floor ≤ R ≤ 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).

5.17.7. 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
```

5.17.8. 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.

5.17.9. 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
```

5.17.10. 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.

5.17.11. 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 -ypl1 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.

5.18. 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.

5.18.1. 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, subsystems 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 different 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 *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.

5.18.2. 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.***

5.18.8. 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 appropriate 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.

Chapter 6

DC — An Interactive Desk Calculator

Dc is an interactive desk calculator program implemented on the UNIX system to do arbitrary-precision integer arithmetic. *Dc* works like a stacking calculator using reverse Polish notation. 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.

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.

6.1. 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 newline characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

6.1.1. number — Push Number onto the Stack

The value of ***number*** is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A through F which are treated as digits with values 10 through 15, respectively, and possible a decimal point. The number may be preceded by an underscore to input a negative number.

*6.1.2. 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.

6.1.3. 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 single character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or newline, is a valid register name.

6.1.4. 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.

6.1.5. d — Duplicate the Top of Stack

The top value on the stack is duplicated.

6.1.6. p — Display the Value on the Top of Stack

Display the value on the top of the stack. The top value remains unchanged.

6.1.7. f — Display All Register and Stack Values

Display all values on the stack and in registers.

6.1.8. x — Execute the Top of Stack

Treat the top element of the stack as a character string, remove it from the stack, and execute it as a string of *dc* commands.

6.1.9. [...] — *Put Character String on Top of Stack*

Put the bracketed character string onto the top of the stack.

6.1.10. q — *Quit From DC*

Exit the program. If executing a string, pop the recursion level by two. If q is capitalized, pop the top value on the stack and pop the string execution level by that value.

6.1.11. Comparison Operators — <*x* >*x* =*x* !<*x* !>*x* !=*x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

6.1.12. v — *Compute Square Root of Top of Stack*

The v command 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.

6.1.13. ! — *Execute a System Command*

This command interprets the rest of the line as a UNIX command. Control returns to *dc* when the UNIX command terminates.

6.1.14. c — *Clear the Stack*

All values on the stack are popped; the stack becomes empty.

6.1.15. 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.

6.1.16. 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.

6.1.17. 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.

6.1.18. z — *Push Value of Stack Level Onto Stack*

The value of the stack level is pushed onto the stack.

6.1.19. ? — *Execute a Line of Input from Input Source*

A line of input is taken from the input source (usually the console) and executed.

6.2. Detailed Description

6.2.1. 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 (centenary 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 (stored at the right end of the string) 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.

6.2.2. 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. When an attempt is made to write beyond the end of a string, the allocator allocates a larger space and then copies the old string into the larger block.

6.2.3. 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 scales of the operands are 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.

6.2.4. 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.

6.2.5. 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.

6.2.6. 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.

6.2.7. 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.

6.2.8. 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

$$x_{n+1} = \frac{1}{2}(x_n + \frac{y}{x_n})$$

The initial guess is found by taking the integer square root of the top two digits.

6.2.9. 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.

6.2.10. 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 `I` command pushes the value of the input base on the stack.

6.2.11. Output Commands

The `p` command displays the top of the stack. 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.

6.2.12. 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.

6.2.13. 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.

6.2.14. 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 `z` pushes the stack size on the stack. The command `X` replaces the number on the top of the stack with its scale factor. The command `Z` replaces the top of the stack with its length.

6.2.15. Subroutine Definitions and Calls

Enclosing an ASCII string in `[]` pushes it onto the stack. The `q` command quits or in executing a string, pops the recursion level by two.

6.2.16. 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 a 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 display the numbers 0-9,

```
[lip1+  si  li1O>a]sa
Osi  lax
```

6.2.17. 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.

6.2.18. 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.

6.3. Design Choices

The real reason for the use of a dynamic storage allocator was that a general-purpose one 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.

6.4. 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).

Chapter 7

BC — Arbitrary-Precision Desk Calculator

Bc is a language and a compiler for doing arbitrary-precision arithmetic on the UNIX 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 memory is exhausted.

The *bc* 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 the *bc* compiler are:

- computation with large integers,
- computation accurate to many decimal places,
- conversion of numbers from one base to another base.

The *bc* 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 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.

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.

7.1. 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
```

bc 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
```

7.2. 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 (upper-case only) 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.

7.3. 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
```

displays the current value of 'scale'.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' is 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.

7.4. 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, it is 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 display the result 21.98, and the line

```
x = a(a(3,4),5)
```

would assign the value 60 to the register *x*.

7.5. 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.

7.6. Control Statements

The `if`, the `while`, and the `for` statements may be used to alter the flow within programs or to iterate. 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(expression-1; relation; expression-2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression-1; relation; expression-2){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 executes its range if and only if the relation is true. Then control passes to the next statement in sequence.

The `while` statement executes 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 prints 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)
```

prints *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
        }
}
```

7.7. 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]
```

assigns a value 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 section 7.9.6 or the C manual [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 these 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.

7.8. 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 loads 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)'). The library sets the scale to 20. You can reset it to something else if you like.

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.

7.9. Notation

In the following pages syntactic categories are in ***italics***; literals are in **boldface**; material in brackets [] is optional.

7.9.1. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

7.9.2. Comments

Comments are introduced by the characters /* and terminated by */.

7.9.3. 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 one-dimensional 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.

7.9.4. Keywords

The following are reserved keywords:

```
ibase   if      obase   break
scale   define  sqrt    auto
length  return  while   quit    for
```

7.9.5. 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.

7.9.6. 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.

7.9.6.1. 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.

7.9.6.2. 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(`*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(`*expression*`)`

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

`scale(`*expression*`)`

The result is the scale of the expression. The scale of the result is zero.

7.9.6.3. Constants

Constants are primitive expressions.

7.9.6.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

7.9.6.5. 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.

7.9.6.6. Binary Operators

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:

$\min(a \times b, \max(\mathbf{scale}, a))$

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:

$\min(a+b, \max(\mathbf{scale}, a, b))$

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**

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.

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.

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*

7.10. 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.

7.11. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

7.11.0.1. 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.

7.11.0.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

7.11.0.3. Quoted string statements

```
"any string"
```

This statement prints the string inside the quotes.

7.11.0.4. If statements

if (*relation*)*statement*

The substatement is executed if the relation is true.

7.11.0.5. While statements

while (*relation*)*statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

7.11.0.6. 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.

7.11.0.7. Break statements

`break`

`break` terminates a `for` or `while` statement.

7.11.0.8. Auto statements

`auto` *identifier*[, *identifier*]

`auto` pushes down the values of the identifiers. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. `auto` must be the first statement in a function definition.

7.11.0.9. Define statements

`define` ([*parameter*[, *parameter*...]]) {*statements*}

`define` defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

7.11.0.10. Return statements

`return`

`return` (*expression*)

`return` terminates of a function, pops 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.

7.11.0.11. Quit

`quit` stops execution of a *bc* program and returns control to UNIX when it is first encountered. Because `quit` is not treated as an executable statement, it cannot be used in a function definition or in an `if`, `for`, or `while` statement.

7.12. Acknowledgement and References

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

[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.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978 — this paper appears in this manual.

[5] R. Morris and L. L. Cherry, *DC — An Interactive Desk Calculator* — this paper appears in the previous chapter of this manual.

Chapter 8

M4 — A Macro Processor

M4 is a macro processor whose 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 higher-level 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

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 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-in macros 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.

8.1. Using the M4 Command

The basic *m4* command line looks like this:

```
tutorial% m4 [ filename ... ]
```

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 to the standard output, which

may be captured for subsequent processing by redirecting the standard output:

```
tutorial% m4 [ filename ... ] > outputfile
```

8.2. Defining Macros

The primary built-in function of *m4* is `define`, which is used to define new macros. The input

```
define(name, stuff)
```

defines the string *name* as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*, unless *name* is redefined, or until *name* is undefined. *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 parenthesis following it.

M4 divides its input into tokens, so 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 that changing *N* does not change *M*.

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 while 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 in turn by its value).

8.3. Quoting and Comments

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)
```

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* pre-defines the name *unix*.

`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)
```

Don't forget the quotes around the argument.

Comments in *m4* are introduced by the # (sharp) character. All text from the # to the end of the line is taken as a comment and otherwise ignored.

8.4. 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* is 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)
```

evaluates to

```
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 commas can be nested inside parentheses. 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.

8.5. 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)')
```

which defines *N1* 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.

`eval` provides the operators (in decreasing order of precedence)

Operator	*Meaning*
`unary + and -`	add and subtract
`** or ^`	exponentiation
`* / %`	multiply, divide, and modulus
`+ -`	binary add and subtract
`== != < <= > >=`	equal, not equal, less than, lessthan or equal, greater than, greater than or equal
`!`	logical 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.

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.

8.6. 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, 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.

8.7. Running System Commands

You can run any UNIX program with the `syscmd` built-in. For example,

```
syscmd(date)
```

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.

8.8. Conditionals

There is a built-in called `ifelse` which enables you to perform arbitrary conditional testing. In its 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" according to whether 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.

8.9. 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)
```

evaluates to

```
ow is the time
```

If either *i* or *n* is 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 in *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 in *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
```

8.10. Printing

The built-in `errprint` writes its arguments to 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!

8.11. Summary of Built-in Macros

```
changequote(L, R)
define(name, replacement)
divert(number)
divnum
dnl
dumpdef(`name', `name', ...)
errprint(s, s, ...)
eval(numeric expression)
ifdef(`name', this if true, this if false)
ifelse(a, b, c, d)
include(file)
incr(number)
index(s1, s2)
len(string)
maketemp(...XXXXX...)
sinclude(file)
substr(string, position, number)
syscmd(s)
```

```
translit(str, from, to)
undefine(`name´)
undivert(number,number,...)
```

8.12. Acknowledgements and References

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.

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.

[1] B. W. Kernighan and P. J. Plauger, *Software* Tools Addison-Wesley, Inc., 1976.

Chapter 9

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 into tokens 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 programmer 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 is then backed up to the end of the current partition, so that the programmer 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*, which is 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 programmer 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 programmer 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 programmer 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 programmer's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the programmer's freedom to write actions is unimpaired. This avoids forcing the programmer 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 programmer. Compatible run-time libraries for the different host languages are also provided. This makes ***lex*** adaptable to different environments and different programmer. Each application may be directed to the combination of hardware and host language appropriate to the task, the programmer´s background, and the properties of local implementations.

Lex turns the programmer´s expressions and actions (called ***source*** in this document) into the host general-purpose language; the generated program is named ***yylex***. The ***yylex*** program recognizes expressions in a stream (called ***input*** in this document) and performs the specified actions for each expression as it is detected — see Figure 1 below.

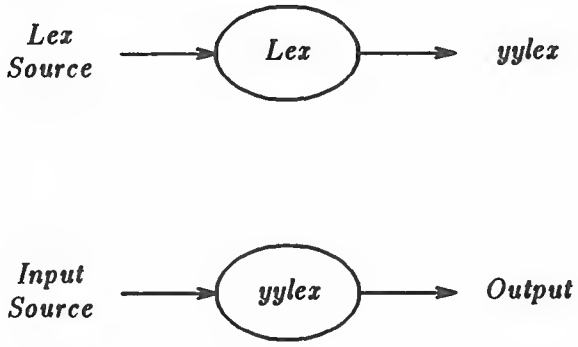


Figure 9-1: An overview of Lex

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 convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates `one or more ...´; and the $ indicates `end-of-line´. No action is specified, so the program generated by ***lex*** (***yylex***) ignores these characters. Everything else is copied to the output stream. 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 scans 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 ends 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*.

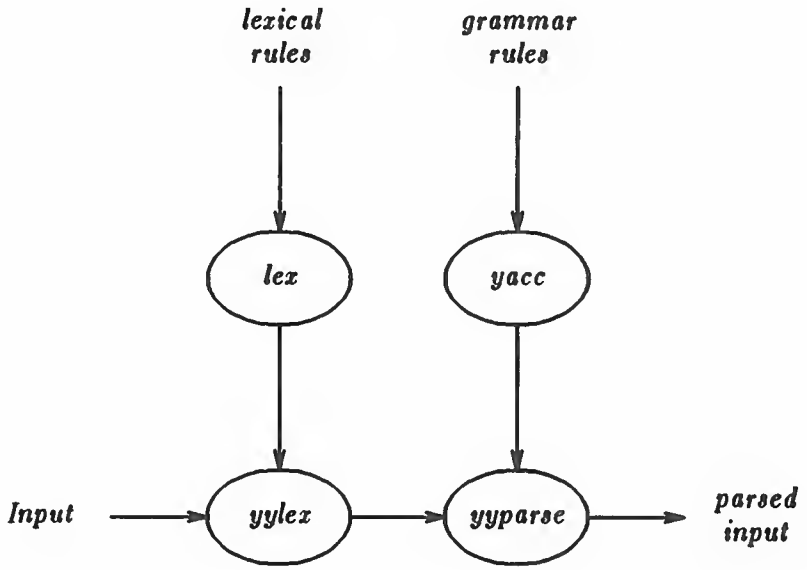


Figure 9-2: Lex with Yacc

Yacc programmers 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 programmer'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 programmer 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* recognizes *ab* and leave the input pointer just before "*cd...*" Such backup is more costly than processing simpler languages.

9.1. Lex Source

The general format of *lex* source is:

```
{definitions}
%%
{rules}
%%
{programmer subroutines}
```

where the definitions and the programmer 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 programmer's control decisions; they are a table, in which the left column contains *regular expressions* (see Section 2) 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 is described later.

9.2. Lex Regular Expressions

The definitions of regular expressions are very similar to those in the UNIX editors *ex*(1) and *vi*(1)[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 must 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 programmer 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 generates 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 system's character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; and

```
[^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

(period) is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive *a* characters, including zero; while

```
a+
```

is one or more instances of *a*. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

Context sensitivity. Lex recognizes a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression is only be matched at the beginning of a line 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 is 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 it is followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in *lex* by *start conditions* as explained in section 9.9 — *Left Context-Sensitivity*. 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 below.

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.

9.3. 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* programmer 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 does this. A frequent rule is

```
[ \t\n] ;
```

which ignores the three spacing characters (blank, tab, and newline).

Another easy way to avoid writing actions is the action character |, which indicates that the action to be used for this rule is the action given for the next rule. The previous example could also have been written

```
" "             |
"\t"            |
"\n"            ;
```

with the same result. The quotes around \n and \t are not required.

In more complex actions, the programmer often wants 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);
```

prints 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 normally matches 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 programmer 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 to be retained in *yytext*. 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 programmer processing
        }
```

which, when faced with a string such as *"abc\"def"* first matches the five characters *"abc*; then the call to *yymore()* tacks the next part of the string, *"def*, onto 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 problem of resolving (in old-style C) 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 = ...
        }
```

performs 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 programmer 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 transmit input or output 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 Section 10 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 programmer 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 programmer 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.

9.4. 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 matches

```
'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, stops after ***'first'***. The consequences of errors like this are mitigated by the fact that the . operator does 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 programmer would like to override this choice. The action REJECT means `go do the next alternative.´ It executes whatever rule was second choice after the current rule. The position of the input pointer is adjusted accordingly. Suppose the programmer 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 is then counted. In this example, of course, the programmer 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 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.

9.5. Lex Source Definitions

Remember the format of the ***lex*** source:

```
{definitions}
%%
{rules}
%%
{programmer routines}
```

So far only the rules have been described. The programmer 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 is 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 the delimiters %{ 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 associates the string given as a translation 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 invoked 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*´.

9.6. Using *lex*

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 in section 3 of the *System Interface Manual for the Sun Workstation*.

The *lex* 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. *Lex* has several options which are described in the *lex*(1) manual page.

9.7. 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 in the *lex* library calls this routine, but if *yacc* is loaded, and its main program is used, *yacc* calls *yylex()*.

In this case each *lex* rule should end with

```
return(token);
```

to return the appropriate token value.

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:

```
tutorial% yacc good
tutorial% lex better
tutorial% cc y.tab.c -ll
tutorial%
```

The *lex* and *yacc* programs can be generated in either order.

9.8. Examples

As a trivial problem, consider copying an input file while adding 3 to every non-negative 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 are 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 constructs a histogram of the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
l 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 then adds *'e'–'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial ***d***. By using the array ***yytext*** the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}     |
{d}{c}{o}{s}     |
{d}{s}{q}{r}{t}  |
{d}{a}{t}{a}{n}  |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}     |
{d}{l}{o}{g}10   |
{d}{m}{i}{n}1    |
{d}{m}{a}{x}1    {
        yytext[0] =+ 'a' - 'd';
        ECHO;
        }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}        {yytext[0] =+ 'r'  - 'd';
                ECHO;
                }
```

To avoid such names as ***dsinx*** being detected as instances of ***dsin***, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+  |
\n      |
.       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

9.9. 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 programmer'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 is 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 begins with the letter ***a***, changing ***magic*** to ***second*** on every line which begins with the letter ***b***, and changing ***magic*** to ***third*** on every line which begins 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;
```

which resets to 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 programmer's code.

9.10. 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 in 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 two 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 9-3: 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.

9.11. Summary of Source Format

The general form of a *lex* source file is:

```
{definitions}
%%
{rules}
%%
{programmer 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 condition declarations, 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:

Operator	*Meaning*
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
{xx}	the translation of xx from the definitions section
x{m,n}	*m* through *n* occurrences of x

9.12. Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic automata; 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 is executed, the programmer must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the programmer's ability to manipulate the not-yet-processed input.

9.13. Acknowledgments and References

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.

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, Murray Hill.

4. A. V. Aho and M. J. Corasick, ***Efficient String Matching: An Aid to Bibliographic Search,*** Comm. ACM **18**, 333-340 (1975).

5. See the papers on ***ex*** and ***vi*** in ***Editing and Text Processing on the Sun Workstation.***

6. M. E. Lesk, ***The Portable C Library,*** Computing Science Technical Report No. 31, Murray Hill.

Chapter 10

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* programmer specifies the structure of the input, together with code to be invoked as each item 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 programmer´s application handled by this subroutine.

The input subroutine produced by *yacc* calls a programmer-supplied routine to return the next basic input item. Thus, the programmer can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The programmer-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* programmer 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 programmer-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 programmer 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 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 — implying 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 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 are 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 *eqn*, a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a *yacc* specification; Section 10.1 describes the preparation of grammar rules, Section 10.2 the preparation of the programmer-supplied actions associated with these rules, and Section 10.3 the preparation of lexical analyzers. Section 10.4 describes the operation of the parser. Section 10.5 discusses various reasons why *yacc* may be unable to produce a parser from a specification, and what to do about it. Section 10.6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 10.7 discusses error detection and recovery. Section 10.8 discusses the operating environment and special features of the parsers *yacc* produces. Section 10.9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10.10 discusses some advanced topics. Section 10.11 has a brief example, and section 10.12 gives a summary of the *yacc* input syntax. Section 10.13 gives an example using some of the more advanced features of *yacc*, and, finally, section 10.14 describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of *yacc*.

10.1. 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
```

Spaces (also called 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 programmer-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´.

10.2. Actions

With each grammar rule, the programmer 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 dollar sign symbol `$´ 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 $1 (the first element in it). 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. The parse tree can be built by supplying actions such as:

```
expr    :       expr  '+'  expr
                        {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The programmer 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 programmer 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.

10.3. Lexical Analysis

The programmer 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 programmer. 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 the 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 `if` or `while` as token names 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 programmer. 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 programmer; 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] and described in the previous chapter of this manual. These lexical analyzers are designed to work in close harmony with ***yacc*** parsers. The specifications 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.

10.4. 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 can read and remember 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 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 bound. 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
```

which pushes state 20 onto the stack, and becomes 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 the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of programmer-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 programmer's 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 `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 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*, and so on. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

10.5. 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 unambiguously 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 programmer 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 (*condition-1*) IF (*condition-2*) *statement-1* ELSE *statement-2*

can be structured according to these rules in two ways:

```
IF  (  condition-1  )  {
          IF  (  condition-2  )  statement-1
}
ELSE  statement-2
```

or

```
IF  (  condition-1  )  {
          IF  (  condition-2  )  statement-1
          ELSE  statement-2
}
```

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 (*condition-1*) IF (*condition-2*) *statement-1*

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

IF (*condition-1*) stat

and then read the remaining input,

ELSE *statement-2*

and reduce

IF (*condition-1*) stat ELSE *statement-2*

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, *statement*-2 read, and then the right hand portion of

IF (*condition-1*) IF (*condition-2*) *statement-1* ELSE *statement-2*

can be reduced by the if-else rule to get

IF (*condition-1*) 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  (  condition-1  )  IF  (  condition-2  )  statement-1
```

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 states, there will be at most one reduce action possible in the state, and this will be the default command. Programmers who encounter 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 programmer 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.

10.6. 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 programmer 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 `.LT.` operator 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 changes 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.

10.7. 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 programmer 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 programmer, 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 programmer can get control to deal with the error actions required by other portions of the program.

10.8. The Yacc Environment

When the programmer 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 name may differ from installation to installation). *Yacc* produces an integer-valued function called `yyparse`. When `yyparse` is called, it in turn repeatedly calls `yylex` — the lexical analyzer supplied by the programmer (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 programmer 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.

The programmer must supply these two routines in one form or another. They can be as simple as the following example, or they can be as complex as needed.

```
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.

The external integer variable `yydebug` is normally set to 0. If it is set to a nonzero value, the parser generates 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.

10.9. 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.

10.9.1. 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 added easily.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in section 10.11 is written following this style, as are the examples in the text of this paper (where space permits). The programmer 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.

10.9.2. 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 programmer 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!

10.9.3. 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.

10.9.4. Reserved Words

Some programming languages permit the programmer 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 programmer can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be ***reserved***; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10.10. Advanced Topics

This section discusses a number of advanced features of ***yacc***.

10.10.1. 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 makes `yyparse` return the value 0; YYERROR makes the parser behave as if the current input symbol results in 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.

10.10.2. 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.

10.10.3. 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 programmer declares the union, and associates a union member name to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, ***yacc*** automatically inserts the appropriate union name, so that no unwanted conversions will take place. In addition, type-checking commands such as ***lint***(1) 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 programmer 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* cannot easily determine the type.

To declare the union, the programmer 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 tag any reference to values returned by these two tokens 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 10.13. 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. This paper is reprinted in this manual.

10.11. 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  */

intc;

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);
}
```

10.12. Yacc Input Syntax

This section describes 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 */
%tokenIDENTIFIER/*   includes identifiers and literals */
%tokenC_IDENTIFIER/*   identifier (but not literal) followed by colon   */
%tokenNUMBER/*    [0-9]+    */

/* reserved words:    %type => TYPE, %left => LEFT, etc. */

%tokenLEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%tokenMARK/* the %% mark */
%tokenLCURL/* the %{ mark */
%tokenRCURL/* the %} mark */

/* ascii character literals stand for themselves */

%startspec

%%

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 ';'
;
```

10.13. An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in section 10.11 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 section 10.11 — 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'*' '/'
%leftUMINUS /* 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 ) );
}
```

10.14. Old Features Supported but not Encouraged

This section 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.

10.15. Acknowledgements and References

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.

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).

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) — in ***Editing and Text Processing on the Sun Workstation***.

8. M.E. Lesk, `Lex — A Lexical Analyzer Generator,' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975) — appears in this manual.