Chapter 7
The Shell

7.1 Introduction

When users log into XENIX, they communicate with the shell command interpreter, sh. This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one function: to read and execute commands from its standard input.

Because the shell gives the user a high-level language in which to communicate with the operating system, XENIX can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With XENIX and the shell, commands can be:

- Combined to form new commands
- Passed positional parameters
- Added or renamed by the user
- Executed within loops or executed conditionally
- Created for local execution without fear of name conflict with other user commands
- Executed in the background without interrupting a session at a terminal

Furthermore, commands can "redirect" command input from one source to another and redirect command output to a file, terminal, printer, or to another command. This provides flexibility in tailoring a task for a particular purpose.

7.2 Basic Concepts

The shell itself (i.e., the program that reads your commands when you log in or that is invoked with the sh command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

7.2.1 How Shells Are Created

In XENIX, a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and may even start (or "fork") new processes. Thus, at any given moment several processes may be executing, some of which are "children" of other processes.

Users log into the operating system and are assigned a "shell" from which they execute. This shell is a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In the XENIX multitasking environment, files may be created in one phase and then sent off to be processed in the "background." This allows the user to

continue working while programs are running.

7.2.2 Commands

The most common way of using the shell is by typing simple commands at your keyboard. A *simple command* is any sequence of arguments separated by spaces or tabs. The first argument (numbered zero) specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. For example, the following command line might be typed to request printing of the files *allan*, *barry*, and *calvin*:

```
lpr allan barry calvin
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell, as parent, creates a child process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a *subshell*) to read the file and execute the commands inside it.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This provides uniformity of invocation.

7.2.3 How the Shell Finds Commands

The shell normally searches for commands in three distinct locations in the file system. The shell attempts to use the command name as given; if this fails, it prepends the string */bin* to the name. If the latter is unsuccessful, it prepends */usr/bin* to the command name. The effect is to search, in order, the current directory, then the directory */bin*, and finally, */usr/bin*. For example, the pr and man commands are actually the files */bin/pr* and */usr/bin/man*, respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If a given command name begins with a slash (/) (e.g., */bin/sort* or */cmd*), the prepending is not performed. Instead, a single attempt is made to execute the command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. The particular sequence of directories

searched may be changed by resetting the shell PATH variable. (Shell variables are discussed later in this chapter).

7.2.4 Generation of Argument Lists

The arguments to commands are very often filenames. Sometimes, these filenames have similar, but not identical, names. To take advantage of this similarity in names, the shell lets the user specify patterns that match the filenames in a directory. If a pattern is matched by one or more filenames in a directory, then those filenames are automatically generated by the shell as arguments to the command.

Most characters in such a pattern match themselves, but there are also XENIX special characters that may be included in a pattern. These special characters are: the star (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters. Inside brackets, a pair of characters separated by a dash (-) matches any character within the range of that pair. Thus [a-de] is equivalent to [abcde].

Examples of metacharacter usage:

*	(*Matches all names in the current directory*)
temp	(*Matches all names containing "temp"*)
[a-f]*	(*Matches all names beginning with "a" through "f"*)
*.c	(*Matches all names ending in ".c"*)
/usr/bin/?	(*Matches all single-character names in /usr/bin*)

This pattern-matching capability saves typing and, more importantly, makes it possible to organize information in large collections of files that are named in a structured fashion, using common characters or extensions to identify related files.

Pattern matching has some restrictions. If the first character of a filename is a period (.), it can be matched only by an argument that literally begins with a period. If a pattern does not match any filenames, then the pattern itself is printed out as the result of the match.

Note that directory names should not contain any of the following characters:

```
* ? [ ]
```

If these characters are used, then infinite recursion may occur during pattern matching attempts.

7.2.5 Quoting Mechanisms

The characters <, >, *, ?, [and] have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. This is done by using single quotation marks (') or double quotation marks (") to surround a string. A backslash (\) before a single character provides this function. (Back quotation marks (`) are used only for command substitution in the shell and do not hide the special meanings of any characters.)

All characters within single quotation marks are taken literally. Thus

```
echostuff='echo $? $*; ls * | wc'
```

results in the string

```
echo $? $*; ls * | wc
```

being assigned to the variable *echostuff*, but it does *not* result in any other commands being executed.

Within double quotation marks, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign ($), the backslash (\), the single quotation mark ('), and the double quotation mark (") itself. Thus, within double quotation marks, variables are expanded and command substitution takes place (both topics are discussed in later sections). However, any commands in a command substitution are unaffected by double quotation marks, so that characters such as star (*) retain their special meaning.

To hide the special meaning of the dollar sign ($) and single and double quotation marks within double quotation marks, precede these characters with a backslash (\). Outside of double quotation marks, preceding a character with a backslash is equivalent to placing single quotation marks around that character. A backslash (\) followed by a newline causes that newline to be ignored and is equivalent to a space. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

Some examples of quoting are shown below:

Input	Shell interprets as:
' ` '	The back quotation mark (`)
'" '	The double quotation mark (")
'`echo one` '	the one word " `echo one`
" \" "	The double quotation mark (")
" `echo one`"	the one word "one"
" `"	illegal (expects another `)
one two	the two words "one" & "two"
"one two"	the one word "one two"
'one two'	the one word "one two"
'one * two'	the one word "one * two"
"one * two"	the one word "one * two"
`echo one`	the one word "one"

7.3 Redirecting Input and Output

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

7.3.1 Standard Input and Output

When a command begins execution, it usually expects that three files are already open: a "standard input", a "standard output", and a "diagnostic output", (also called "standard error"). A number called a *file descriptor* is associated with each of these files. By convention, file descriptor 0 is associated with the standard input, file descriptor 1 with the standard output, and file descriptor 2 with the diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the terminal screen). The shell permits the files to be redirected elsewhere before control is passed to an invoked command.

An argument to the shell of the form "<*file*" or ">*file*" opens the specified file as the standard input or output (in the case of output, destroying the previous contents of *file*, if any). An argument of the form ">>*file*" directs the standard output to the end of *file*, thus providing a way to append data to the file without destroying its existing contents. In either of the two output cases,

the shell creates *file* if it does not already exist. Thus

```
>output
```

alone on a line creates a zero-length file. The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of filenames occurs after these substitutions. This means that

```
echo 'this is a test' > *.gal
```

produces a one-line file named **.gal*. Similarly, an error message is produced by the following command, unless you have a file with the name "?":

```
cat < ?
```

So remember, special characters are *not* expanded in redirection arguments. The reason this is so is that redirection arguments are scanned by the shell *before* pattern recognition and expansion takes place.

7.3.2 Diagnostic and Other Outputs

Diagnostic output from XENIX commands is normally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines.) You can redirect this error output to a file by immediately prepending the number of the file descriptor (2 in this case) to either output redirection symbol (> or >>). The following line appends error messages from the cc command to the file named *ERRORS*:

```
cc testfile.c 2>>ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening spaces or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if *cmd* puts output on file descriptor 9, then the following line will direct that output to the file *savedata*:

```
cmd 9>savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect

independently all the different outputs. Suppose, for example, that *cmd* directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard  2>error  9>data
```

7.3.3 Command Lines and Pipelines

A sequence of commands separated by the vertical bar (|) makes up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by *pipes*, that is, the output of each command (except the last one) becomes the input of the next command in line.

A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; **sort** is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline:

```
nroff -mm text | col | lpr
```

Nroff is a text formatter available in the XENIX Text Processing System whose output may contain reverse line motions, **col** converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and **lpr** does the actual printing. The flag **-mm** indicates one of the commonly used formatting options, and *text* is the name of the file to be formatted.

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

- **who**
  Prints the list of logged-in users on the terminal screen.

- **who >>log**
  Appends the list of logged-in users to the end of file *log*.

- **who | wc -l**
  Prints the number of logged-in users. (The argument to **wc** is pronounced "minus ell".)

- who | pr
  Prints a paginated list of logged-in users.

- who | sort
  Prints an alphabetized list of logged-in users.

- who | grep bob
  Prints the list of logged-in users whose login names contain the string *bob*.

- who | grep bob | sort | pr
  Prints an alphabetized, paginated list of logged-in users whose login names contain the string *bob*.

- { date; who | wc -l; } >>log
  Appends (to file *log*) the current date followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.

- who | sed -e 's/ .*//' | sort | uniq -d
  Prints only the login names of all users who are logged in more than once. Note the use of sed as a filter to remove characters trailing the login name from each line. (The ".*" in the sed command is preceded by a space.)

The who command does not *by itself* provide options to yield all these results—they are obtained by combining who with other commands. Note that who just serves as the data source in these examples. As an exercise, replace "who |" with "</etc/passwd" in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line, even at the start. This means that

```
<infile >outfile sort|pr
```

is the same as

```
sort|pr <infile >outfile
```

7.3.4 Command Substitution

Any command line can be placed within back quotation marks (`...`) so that the output of the command replaces the quoted command line itself. This concept is known as *command substitution*. The command or commands enclosed between back quotation marks are first executed by the shell and then their output replaces the whole expression, back quotation marks and all. This feature is often used to assign to shell variables. (Shell variables are described in the next section.) For example,

```
today=`date`
```

assigns the string representing the current date to the variable "today"; for example "Tue Nov 27 16:01:09 EST 1982". The following command saves the number of logged-in users in the shell variable *users*:

```
users=`who | wc -l`
```

Any command that writes to the standard output can be enclosed in back quotation marks. Back quotation marks may be nested, but the inside sets must be escaped with backslashes (\). For example:

```
logmsg=`echo Your login directory is \`pwd\``
```

will display the line "your login directory is *name of login directory*". Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example,

```
read first init last
```

takes an input line of the form

```
G. A. Snyder
```

and has the same effect as typing:

```
first=G.  init=A.  last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

7.4 Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as *positional parameters*; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

7.4.1 Positional Parameters

When a shell procedure is invoked, the shell implicitly creates *positional parameters*. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter $0. The first command argument is called $1, and so on. The shift command may be used to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files:

```
while test '$1'
        do case $1 in
        -a) A=aoption ; shift ;;
        -b) B=boption ; shift ;;
        -c) C=coption ; shift ;;
        -*) echo "bad option" ; exit 1 ;;
        *) process rest of files
        esac
done
```

One can explicitly force values into these positional parameters by using the set command. For example,

```
set abc def ghi
```

assigns the string "abc" to the first positional parameter, $1, the string "def" to $2, and the string "ghi" to $3. Note that $0 may not be assigned a value in this way—it always refers to the name of the shell procedure; or in the login shell, to the name of the shell.

7.4.2 User-Defined Variables

The shell also recognizes alphanumeric variables to which string values may be assigned. A simple assignment has the syntax:

name=string

Thereafter, $*name* will yield the value *string*. A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. No spaces surround the equal sign (=) in an assignment statement. Note that positional parameters may not appear on the left side of an assignment statement; they can only be set as described in the previous section.

More than one assignment may appear in an assignment statement, but beware: *the shell performs the assignments from right to left.* Thus, the following command line results in the variable "A" acquiring the value "abc":

```
A=$B B=abc
```

The following are examples of simple assignments. Double quotation marks around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, while also allowing variable substitution (also known as "parameter substitution") to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign ($) are replaced by the corresponding values, if any. Single quotation marks inhibit variable substitution:

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable "echovar" has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string "echo". No quotation marks are needed around the string of asterisks being assigned to *stars* because pattern matching (expansion of star, the question mark, and brackets) does not apply in this context. Note that the value of *$asterisks* is the literal string "$stars", *not* the string "*****", because the single quotation marks inhibit substitution.

In assignments, spaces are not re-interpreted after variable substitution, so that the following example results in *$first* and *$second* having the same value:

```
first=`a string with embedded spaces`
second=$first
```

In accessing the values of variables, you may enclose the variable name in braces {...} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input:

```
a=`This is a string`
echo "${a}ent test of variables."
```

Here, the echo command prints:

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for "$aent" and print:

```
test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user:

HOME Initialized by the login program to the name of the user's *login directory*, that is, the directory that becomes the current directory upon completion of a login; cd without arguments switches to the $HOME directory. Using this variable helps keep full pathnames out of shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

IFS The variable that specifies which characters are *internal field separators*. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set IFS to include that delimiter.) The shell initially sets IFS to include the blank, tab, and newline characters.

MAIL The pathname of a file where your mail is deposited. If MAIL is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL must be set by the user and "exported". (The export command is discussed later in this chapter.) (The presence of mail in the standard mail file is also announced at login, regardless of whether MAIL is set.)

PATH The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes PATH to the list *:/bin:/usr/bin* where a null argument appears in front of the first colon. A null anywhere in the path list represents the current directory. On some systems, a search of the current directory is *not* the default and the PATH variable is initialized instead to */bin:/usr/bin*. If you wish to search your current directory last, rather than first, use:

```
PATH=/bin:/usr/bin::
```

Here, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, *$HOME/bin*) and cause it to be searched *before* the other three directories by using:

```
PATH=$HOME/bin::/bin:/usr/bin
```

"PATH" is normally set in your *.profile* file.

PS1 The variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is "$ " (a

dollar sign ($) followed by a blank).

PS2 The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2. The default value for this variable is "> " (a greater-than symbol followed by a space).

In general, you should be sure to export all of the above variables so that their values are passed to all shells created from your login. Use export at the end of your *.profile* file. An example of an export statement follows:

```
export HOME IFS MAIL PATH PS1 PS2
```

7.4.3 Predefined Special Variables

Several variables have special meanings; the following are set *only* by the shell:

$# Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, $# yields the number of the highest set positional parameter. Thus

```
sh cmd a b c
```

automatically sets $# to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
        echo 'two or more args required'; exit
fi
```

$? Contains the exit status of the last command executed (also referred to as "return code", "exit code", or "value"). Its value is a decimal string. Most XENIX commands return zero to indicate successful completion. The shell itself returns the current value of $? as its exit status.

$$ The process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files. XENIX provides no mechanism for the automatic creation and deletion of temporary files; a file exists until it is explicitly removed. Temporary files are generally undesirable objects; the XENIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files; note that the directories */usr* and */usr/tmp*

are cleared out if the system is rebooted.

```
#               use current process id
#               to form unique temp file
temp=/usr/temp/$$
ls > $temp
#               commands here, some of which use $temp
rm $temp
#               clean up at end
```

$! The process number of the last process run in the background (using the ampersand (&)). This is a string containing from one to five digits.

$-** A string consisting of names of execution flags currently turned on in the shell. For example, **$- might have the value "xv" if you are tracing your output.

7.5 The Shell State

The state of a given instance of the shell includes the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory.

The state of a shell may be altered in various ways. These include changing the working directory with the cd command, setting several flags, and by reading commands from the special file, *.profile*, in your login directory.

7.5.1 Changing Directories

The cd command changes the current directory to the one specified as its argument. This can and should be used to change to a convenient place in the directory structure. Note that cd is often placed within parentheses to cause a subshell to change to a different directory and execute some commands without affecting the original shell.

For example, the first sequence below copies the file */etc/passwd* to */usr/you/passwd*; the second example first changes directory to */etc* and then copies the file:

```
cp /etc/passwd /usr/you/bin/passwd
(cd /etc ; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

7.5.2 The .profile File

The file named *.profile* is read each time you log in to XENIX. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from *.profile*, does the shell read commands from the standard input—usually the terminal.

7.5.3 Execution Flags

The set command lets you alter the behavior of the shell by setting certain shell flags. In particular, the –x and –v flags may be useful when invoking the shell as a command from the terminal. The flags -x and -v may be set by typing:

```
set -xv
```

The same flags may be turned *off* by typing:

```
set +xv
```

These two flags have the following meaning:

–v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

–x Commands and their arguments are printed as they are executed. (Shell control commands, such as for, while, etc., are not printed, however.) Note that –x causes a trace of only those commands that are actually executed, whereas –v prints each line of input until a syntax error is detected.

The set command is also used to set these and other flags within shell procedures.

7.6 A Command's Environment

All variables and their associated values that are known to a command at the beginning of its execution make up its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the export command. The export command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
#                keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are *not* counted as arguments to the procedure and do not affect **$#**.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, type:

```
export
```

You will also get a list of variables that have been made **readonly**. To get a list of name-value pairs in the current environment, type either

```
printenv
```

or

```
env
```

7.7 Invoking the Shell

The shell is a command and may be invoked in the same way as any other command:

sh *proc* [*arg*...]	A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated.

sh -v *proc* [*arg*...]	This is equivalent to putting "set -v" at the beginning of *proc*. It can be used in the same way for the -x, -e, -u, and -n flags.
proc [*arg*...]	If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of: sh proc args An advantage of this form is that variables that have been exported in the shell will still be exported from *proc* when this form is used (because the shell only forks to read commands from *proc*). Thus any changes made within *proc* to the values of exported variables will be passed on to subsequent commands invoked from *proc*.

7.8 Passing Arguments to Shell Procedures

When a command line is scanned, any character sequence of the form **$*n*** is replaced by the *n*th argument to the shell, counting the name of the shell procedure itself as **$0**. This notation permits direct reference to the procedure name and to as many as nine positional parameters. Additional arguments can be processed using the **shift** command or by using a **for** loop.

The **shift** command shifts arguments to the left; i.e., the value of **$1** is thrown away, **$2** replaces **$1**, **$3** replaces **$2**, and so on. The highest-numbered positional parameter becomes *unset* (**$0** is never shifted). For example, in the shell procedure *ripple* below, **echo** writes its arguments to the standard output.

```
#               ripple command
while test $# != 0
do
                echo $1 $2 $3 $4 $5 $6 $7 $8 $9
                shift
done
```

Lines that begin with a number sign (#) are comments. The looping command, **while**, is discussed in Section 7.9.3 of this chapter. If the procedure were invoked with

```
ripple a b c
```

it would print:

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

The special shell variable "star" (**$***) causes substitution of all positional parameters except **$0**. Thus, the **echo** line in the *ripple* example above could be written more compactly as:

```
echo $*
```

These two **echo** commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The shell star variable (**$***) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command: For example

```
wc $*
```

counts the words of each of the files named on the command line.

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a newline or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via back quotation marks) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for *internal field separators*, that is, for any characters specified by IFS to break the command line into distinct arguments; *explicit* null arguments (specified by "" or ``) are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then filename generation occurs with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes, command lines are built inside a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command **eval** is available for this purpose. **Eval** takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output='| wc -l'
eval $command $output
```

This segment of code results in the execution of the command line

```
who | wc -l
```

The output of **eval** cannot be redirected. However, uses of **eval** can be nested, so that a command line can be evaluated several times.

7.9 Controlling the Flow of Control

The shell provides several commands that implement a variety of control structures useful in controlling the flow of control in shell procedures. Before describing these structures, a few terms need to be defined.

A *simple command* is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, *not* to the command.

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command but the last is connected (by a *pipe*) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is nonzero if the exit status of either the first or last process in the pipeline is nonzero.

A *command list* is a sequence of one or more pipelines separated by a semicolon (;), an ampersand (&), an "and-if" symbol (&&), or an "or-if" (||) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing INTERRUPT or QUIT. It is also immune to logouts with CNTRL-D. However, CNTRL-D *will* abort the command if you are operating over a dial-up line. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. In the above example without **nohup**, if you log out from a dial-up line while cc is still executing, cc will be killed and your output will disappear.

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

The and-if and or-if (&& and ||) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are lower than the ampersand (&) and the vertical bar (|)). In the command line

```
cmd1 || cmd2
```

the first command, *cmd1,* is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:

```
if          cmd1
            test $? != 0
then
            cmd2
fi
```

The and-if operator (&&) operator yields a complementary test. For example, in the following command line

```
cmd1 && cmd2
```

the second command is executed only if the first *succeeds* (and has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents:

```
{ nroff -mm text1; nroff -mm text2; } | lpr
```

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

7.9.1 Using the if Statement

The shell provides structured conditional capability with the if command. The simplest if command has the following form:

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

The command list following the if is executed and if the last command in the list has a zero exit status, then the command list that follows then is executed. The

word fi indicates the end of the if command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an else clause can be given with the following structure:

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

Multiple tests can be achieved in an if command by using the elif clause, although the case statement (See Section 7.9.2) is better for large numbers of tests. For example:

```
if      test -f "$1"
#                                       is $1 a file?
then    pr $1
elif    test -d "$1"
#                                       else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows: if the value of the first positional parameter is a filename (-f), then print that file; if not, then check to see if it is the name of a directory (-d). If so, change to that directory (cd) and print all the files there (pr*). Otherwise, echo the error message.

The if command may be nested (but be sure to end each one with a fi). The newlines in the above examples of if may be replaced by semicolons.

The exit status of the if command is the exit status of the last command executed in any then clause or else clause. If no such command was executed, if returns a zero exit status.

Note that an alternate notation for the test command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

```
if      [ -f "$1" ]
#                                       is $1 a file?
then    pr $1
elif    [ -d "$1" ]
#                                       else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

Note that a space after the left bracket and one before the right bracket are essential in this form of the syntax.

7.9.2 Using the case Statement

A multiple test conditional is provided by the case command. The basic format of the case statement is:

```
case string in
        pattern ) command-list ;;
        ...
        pattern ) command-list ;;
esac
```

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in filename generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the case and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a star (*) is the first pattern in a case, no other patterns are looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by vertical bars (|).

```
case $i in
        *.c)                    cc $i
                                ;;
        *.h | *.sh)
                                : do nothing
                                ;;
        *)                      echo "$i of unknown type"
                                ;;
esac
```

In the above example, no action is taken for the second set of patterns because the null, colon (:) command is specified. The star (*) is used as a default pattern, because it matches any word.

The exit status of case is the exit status of the last command executed in the case command. If no commands are executed, then case has a zero exit status.

7.9.3 Conditional Looping: while and until

A while command has the general form:

```
while command-list
do
        command-list
done
```

The commands in the first *command-list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second *command-list* are executed. This sequence is repeated as long as the exit status of the first *command-list* is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing while with until.

Any newline in the above example may be replaced by a semicolon. The exit status of a while (or until) command is the exit status of the last command executed in the second *command-list*. If no such command is executed, while (or until) has a zero exit status.

7.9.4 Looping Over a List: for

Often, one wishes to perform some set of operations for each file in a set of files, or execute some command once for each of several arguments. The for command can be used to accomplish this. The for command has the format:

```
for variable in word-list
do
        command-list
done
```

Here *word-list* is a list of strings separated by blanks. The commands in the *command-list* are executed once for each word in the *word-list*. *Variable* takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following for loop causes each of the C source files *xec.c*, *cmd.c*, and *word.c* in the current directory to be compared with a file of the same name in the directory */usr/src/cmd/sh*:

```
for CFILE in xec cmd word
do      diff ${CFILE}.c /usr/src/cmd/sh/${CFILE}.c
done
```

Note that the first occurrence of CFILE immediately after the word for has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the "in *word-list*" part of a for command; this causes the current set of positional parameters to be used in place of word-list. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments. Create a file named *echo2* that contains the following shell script:

```
for word
do echo $word$word
done
```

Give *echo2* execute status:

```
chmod +x echo2
```

Now type the following command:

```
echo2 ma pa bo fi yo no so ta
```

The output from this command is:

```
mama
papa
bobo
fifi
yoyo
nono
soso
tata
```

7.9.5 Loop Control: break and continue

The **break** command can be used to terminate execution of a **while** or a **for** loop. **Continue** requests the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done**. Exit from *n* levels is obtained by **break** *n*.

The **continue** command causes execution to resume at the nearest enclosing **for**, **while**, or **until** statement, i.e., the one that begins the innermost loop containing the **continue**. You can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do              echo "Please enter data"
                read response
                case "$response" in
                "done")                 break
                                        # no more data
                                        ;;
                "")                     # just a carriage return,
                                        # keep on going
                                        continue
                                        ;;
                *)                      # process the data here
                                        ;;
                esac
done
```

7.9.6 End-of-File and exit

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a CNTRL-D which is the same as logging out.

The exit command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing "exit 0" at the end of the file.

7.9.7 Command Grouping: Parentheses and Braces

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command line—they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you type

```
garble(stuff)
```

the shell prints an error message. Quoted lines, such as

```
garble"("stuff")"
"garble(stuff)"
```

are interpreted correctly. Other quoting mechanisms are discussed in section 7.2.3.2, "Quoting Mechanisms".

This capability of creating a subshell by grouping commands is useful when performing operations without affecting the values of variables in the current shell, or when temporarily changing the working directory and executing commands in the new directory without having to return to the current directory.

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n | lpr& ; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n | lpr&)
```

accomplish the same result: a copy of */usr/docs/otherdir/doc.n* is sent to the lineprinter. (Note that nroff is a command available in the XENIX Text Processing System.) However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) may also be used to group commands together. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

7.9.8 Input/Output Redirection and Control Commands

The shell normally does *not* fork and create a new shell when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

1. Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the

effect of using parentheses to group commands).

2. Control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

7.9.9 Transfer to Another File and Back: The Dot (.) Command

A command line of the form

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the *.profile* file with:

```
. .profile
```

7.9.10 Interrupt Handling: trap

Shell procedures can use the trap command to disable a signal (cause it to be ignored), or redefine its action. The form of the trap command is:

trap *arg signal-list*

Here *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers as described in *signal*(S)) in the XENIX *Reference Manual*. The most important of these signals follow:

Number	Signal
00	KILL (CNTRL-U)
01	HANGUP
02	INTERRUPT character
03	QUIT
09	KILL (cannot be caught or ignored)
11	segmentation violation (cannot be caught or ignored)
15	software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the trap command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in

the file *errdirect* when it is interrupted, thus giving the user information as to how much of the job was done:

```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
          # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
(trap "echo `pwd` >errdirect" 2 3 15)
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the trap command as a signal generated by exiting from a shell. This occurs either with an exit command, or by "falling through" to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string (''or ""), then the signals in the signal list are ignored by the shell.

The trap command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
   # commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15 (kill) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotation marks are executed. The exit command must be included, or else the shell continues reading commands where it left off when the signal was received. The "trap 0" in the above procedure turns off the original traps 1, 2, 3, and 15 on exits from the shell, so that the exit command does not reactivate the execution of the trap commands.

Sometimes the shell continues reading commands after executing trap commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands typed at the terminal until an end-of-file (CNTRL-D) or an interrupt is received. An end-of-file causes the read command to return a nonzero exit status, and thus the while loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:

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

Several traps may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the shell implements the trap command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate trap commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned *after* the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the read command, for which traps are serviced immediately, so that read can be interrupted while waiting for input.

7.10 Special Shell Commands

There are several special commands that are *internal* to the shell, some of which have already been mentioned. The shell does not fork to execute these commands, so no additional processes are spawned. These commands should be used whenever possible, because they are, in general, faster and more efficient than other XENIX commands. The trade-off for this efficiency is that redirection of input and output is not allowed for most of these special commands.

Several of the special commands have already been described because they affect the flow of control. They are dot (.), **break**, **continue**, **exit**, and **trap**. The **set** command is also a special command. Descriptions of the remaining special commands are given here:

:	The null command. This command does nothing and can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#) which can be used to insert comments to the end-of-line. Beware: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.
cd *arg*	Make *arg* the current directory. If *arg* is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned. Specifying cd with no *arg* is equivalent to typing "cd $HOME" which takes you to your home directory.
exec *arg*...	If *arg* is a command, then the shell executes the command without forking and returning to the current shell. This effectively a "goto" and no new process is created. Input and output redirection arguments are allowed on the command line. If *only* input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.
newgrp *arg*...	The **newgrp** command is executed, replacing the shell. **Newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any variables that were exported will no longer be marked as such.
read *var*...	One line (up to a newline) is read from the standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.
readonly *var*...	The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all exported variables is given.
times	The accumulated user and system times for processes run from the current shell are printed.

umask *nnn*

The user file creation mask is set to *nnn*. If *nnn* is omitted, then the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal umask of 137 corresponds to the following bit-mask and permission settings for a newly created file:

User	user	group	other
Octal	1	3	7
bit-mask	001	011	111
permissions	rw-	r--	---

See *umask*(C) in the XENIX *Reference Manual* for information on the value of *nnn*.

wait

The shell waits for all currently active child processes to terminate. The exit status of wait is always zero.

7.11 Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the *mode* of the file to make it *executable*, thus permitting it to be invoked by

```
proc args
```

rather than

```
sh proc args
```

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. To set up a simple procedure, first create a file named *mailall* with the following contents:

```
LETTER=$1
shift
for i in $*
do mail $i <$LETTER
done
```

Next type:

```
chmod +x mailall
```

The new command might then be invoked from within the current directory by typing:

```
mailall letter joe bob
```

Here *letter* is the name of the file containing the message you want to send, and *joe* and *bob* are people you want to send the message to. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If *mailall* were thus created in a directory whose name appears in the user's PATH variable, the user could change working directories and still invoke the *mailall* command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the *dot* command (.) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer writing shell procedures to writing C programs. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.

2. A shell procedure has no corresponding object program that must be generated and maintained.

3. A shell procedure is easy to create quickly, use a few times, and then remove.

4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named *bin*. This name is derived from the word "binary", and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their PATH variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it may become difficult to keep track of your environment and efficiency may suffer.

7.12 More About Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

-e This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.

-u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.

-t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.

-n This is a "don't execute" flag. On occasion, one may want to check a procedure for syntax errors, but not execute the commands in the procedure. Using "set -nv" at the beginning of a file will accomplish this.

-k This flag causes all arguments of the form *variable=value* to be treated as keyword parameters. When this flag is *not* set, only such arguments that appear before the command name are treated as keyword parameters.

7.13 Supporting Commands and Features

Shell procedures can make use of any XENIX command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use.

7.13.1 Conditional Evaluation: test

The test command evaluates the expression specified by its arguments and, if the expression is true, test returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. Test also returns a nonzero exit status if it has no arguments. Often it is convenient to use the test command as the first command in the command list following an if or a while. Shell variables used in test expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets may be used as an alias to test, so that

[*expression*]

has the same effect as:

test *expression*

Note that the spaces before and after the *expression* in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

-r *file*	True if the named file exists and is readable by the user.
-w *file*	True if the named file exists and is writable by the user.
-x *file*	True if the named file exists and is executable by the user.
-s *file*	True if the named file exists and has a size greater than zero.
-d *file*	True if the named file is a directory.
-f *file*	True if the named file is an ordinary file.
-z *s1*	True if the length of string *s1* is zero.
-n *s1*	True if the length of the string *s1* is nonzero.
-t *fildes*	True if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by default.
s1 = *s2*	True if strings *s1* and *s2* are identical.
s1 != *s2*	True if strings *s1* and *s2* are *not* identical.
s1	True if *s1* is *not* the null string.
n1 -eq *n2*	True if the integers *n1* and *n2* are algebraically equal; other algebraic comparisons are indicated by -ne (not equal), -gt (greater than), -ge (greater than or equal to), -lt (less than), and -le (less than or equal to).

These may be combined with the following operators:

!	Unary negation operator.
-a	Binary logical AND operator.
-o	Binary logical OR operator; it has lower precedence than the logical AND operator (-a).
(*expr*)	Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to test.

7.13.2 Echoing Arguments

The echo command has the following syntax:

```
echo [ options ] [ args ]
```

Echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. Often, it is used to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command

```
ls
```

is often replaced by

```
echo *
```

because the latter is faster and prints fewer lines of output.

The –n option to echo removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow typing on the same line as the prompt:

```
echo -n 'enter name:'
read name
```

The echo command also recognizes several escape sequences described in *echo*(C) in the XENIX *Reference Manual*.

7.13.3 Expression Evaluation: expr

The expr command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; expr can be used inside grave accents to set a variable. Some typical examples follow:

```
#           increment $A
A=`expr $a + 1`
#           put third through last characters of
#           $1 into substring
substring=`expr "$1" : '..\(.*\)' `
#           obtain length of $1
c=`expr "$1" : '.*' `
```

The most common uses of expr are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

7.13.4 True and False

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might type:

```
while true
        do echo forever
        done
```

This will echo "forever" on the screen until an INTERRUPT is typed.

7.13.5 In-Line Input Documents

Upon seeing a command line of the form

command << *eofstring*

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring*. (By appending a minus (-) to the input redirection symbol (<<), leading spaces and tabs are deleted from each line of the input document before the shell passes the line to *command*.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you may quote any character of *eofstring*:

```
command << \eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could type:

```
cat <<- xx
        This message will be printed on the
        terminal with leading tabs and spaces
        removed.
xx
```

This in-line input document feature is most useful in shell procedures. Note that in-line input documents may not appear within grave accents.

7.13.6 Input/Output Redirection Using File Descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the *write*(S) system call (see the XENIX *Reference Manual*. The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By typing

fd1>&*fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing:

command 2>&1

If you wanted to redirect both standard output and standard error output to the same file, you would type:

command 1>file 2>&1

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard input. You could type

fda<&fdb

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

7.13.7 Conditional Substitution

Normally, the shell replaces occurrences of $*variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in anyone of the following ways:

```
A=
bcd=""
efg=''
set '' ""
```

The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. *Parameter* as used below refers to either a digit or a variable name.

${*variable*:-*string*}** If *variable* is set and is nonnull, then substitute the value **$*variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

${*variable*:=*string*}** If *variable* is set and is nonnull, then substitute the value **$*variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value **$*variable*** in place of this expression. Positional parameters may not be assigned values in this fashion.

${*variable*:?*string*} If *variable* is set and is nonnull, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

variable: *string*

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message

variable: parameter null or not set

is printed instead.

${*variable*:+*string*} If *variable* is set and is nonnull, then substitute *string* for this expression. Otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

   "PATH"=${PATH:-':/bin:/usr/bin'}

   This says, if PATH has ever been set and is not null, then keep its current value; otherwise, set it to the string ":/bin:/usr/bin".

2. This example automatically assigns the HOME variable a value:

   cd ${HOME:='/usr/gas'}

   If HOME is set, and is not null, then change directory to it. Otherwise set HOME to the given value and change directory to it.

7.13.8 Invocation Flags

There are four flags that may be specified on the command line when invoking the shell. These flags may not be turned on with the set command:

-i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.

-s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. The shell you get upon logging into the system has the -s flag turned on.

-c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored. Double quotation marks should be used to enclose a multiword string, in order to allow for variable substitution.

7.14 Effective and Efficient Shell Programming

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the time command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

7.14.1 Number of Processes Generated

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built in:

break	case	cd	continue	eval
exec	exit	export	for	if
newgrp	read	readonly	set	shift
test	times	trap	umask	until
wait	while	.	:	{}

Parentheses, (), are built into the shell, but commands enclosed within them are executed as a child process, i.e., the shell does a fork, but no exec. Any command not in the above list requires both fork and exec.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

processes = (k*n) + c

where *k* and *c* are constants, and *n* may be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero.

Any procedure whose complexity measure includes n^2 terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named *split*, whose text is given below:

```
#       split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
                # read stdin into temp file
                # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
                # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                # lines with only numbers onto $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total - \($end1 - $start1\) \
- \($end2 - $start2\)`"
echo "$total read, $lost thrown away"
```

For each iteration of the loop, there is one expr plus either an echo or another expr. One additional echo is executed at the end. If *n* is the number of lines of input, the number of processes is 2*n + 1.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C. Shell procedures should not be used to scan or build files a character at a time.

7.14.2 Number of Data Bytes Accessed

It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to sort will be much smaller:

```
sort file | grep pattern
grep pattern file | sort
```

7.14.3 Shortening Directory Searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of cd, the change directory command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```
ls -l /usr/bin/* >/dev/null
cd /usr/bin; ls -l * >/dev/null
```

The second command will run faster because of the fewer directory searches.

7.14.4 Directory-Search Order and the PATH Variable

The PATH variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current PATH variable. As an example, consider the effect of invoking **nroff** (i.e., */usr/bin/nroff*) when the value of PATH is ":/bin:/usr/bin". The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to PATH can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin*. Careless PATH setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin::/usr/bin:/usr/john/bin:/usr/localbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the PATH variable inside the procedure so that the fewest possible directories are searched in an optimum order.

7.14.5 Good Ways to Set Up Directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small directory; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 30 or 286 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

7.15 Shell Procedure Examples

The power of the XENIX shell command language is most readily seen by examining how XENIX's many labor-saving utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called "scripts"). Note the use of the number sign (#) to introduce comments into shell procedures.

It is intended that the following steps be carried out for each procedure:

1. Place the procedure in a file with the indicated name.
2. Give the file execute permission with the **chmod** command.
3. Move the file to a directory in which commands are kept, such as your own *bin* directory.
4. Make sure that the path of the *bin* directory is specified in the PATH variable found in *.profile*.
5. Execute the named command.

BINUNIQ

```
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both */bin* and */usr/bin*. It is done because files in */bin* will "override" those in */usr/bin* during most searches and duplicates need to be weeded out. If the */usr/bin* file is obsolete, then space is being wasted; if the */bin* file is outdated by a corresponding entry in */usr/bin* then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of "sort | uniq" to find matches and duplications.

COPYPAIRS

```
#       Usage: copypairs file1 file2 ...
#       Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
        cp $1 $2
        shift; shift
done
if test "$1" != ""
        then echo "$0: odd number of arguments"
fi
```

This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop, because you can adjust the positional parameters with the **shift** command to handle related arguments.

COPYTO

```
#       Usage: copyto dir file ...
#       Copies argument files to "dir",
#       making sure that at least
#       two arguments exist, that "dir" is a directory,
#       and that each additional argument
#       is a readable file.
if test $# -lt 2
        then    echo "$0: usage: copyto directory file ..."
elif test ! -d $1
        then    echo "$0: $1 is not a directory";
else    dir=$1; shift
        for eachfile
        do      cp $eachfile $dir
done
fi
```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to copyto but the first; the original $1 is shifted off.

DISTINCT1

```
#       Usage: distinct1
#       Reads standard input and reports list of
#       alphanumeric strings that differ only in case,
#       giving lowercase form of each.
tr -cs 'A-Za-z0-9' '\012' |sort -u | \
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It may not be immediately obvious how this command works. You may wish to consult *tr*(C), *sort*(C), and *uniq*(C) in the XENIX *Reference Manual* if you are completely unfamiliar with these commands. The tr command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The sort command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next tr converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. The first line below is equivalent to the last two lines, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output.

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

DRAFT

```
#       Usage: draft file(s)
#       Print manual pages for Diablo printer.
for i in $*
        do nroff -man $i | lpr
done
```

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
#       Usage: edfind file arg
#       Finds the last occurrence in "file" of a line
#       whose beginning matches "arg", then prints
#       3 lines (the one before, the line itself,
#       and the one after)
ed - $1 <<-EOF
        ?^$2?
        -,+p
        q
EOF
```

This illustrates the practice of using ed in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
#       Usage: edlast file
#       Prints the last line of file,
#       then deletes that line.
ed - $1 <<-\!
        $p
        $d
        w
        q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation point (!). Variable substitution is prohibited within the input text because of the backslash.

FSPLIT

```
#       Usage: fsplit file1 file2
#       Reads standard input and divides it into 3 parts
#       by appending any line containing at least one letter
#       to file1, appending any line containing digits but
#       no letters to file2, and by throwing the rest away.
count=0 gone=0
while read next
do
        count="`expr $count + 1`"
        case "$next" in
        *[A-Za-z]*)
                echo "$next" >> $1 ;;
        *[0-9]*)
                echo "$next" >> $2 ;;
        *)
                gone="`expr $gone + 1`"
        esac
done
echo "$count lines read, $gone thrown away"
```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when read encounters an end-of-file. Note the use of the expr command.

Don't use the shell to read a line at a time unless you must—it can be an extremely slow process.

LISTFIELDS

```
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a colon on a newline. Thus, if given the following input

```
joe newman: 13509 NE 78th St: Redmond, Wa 98062
```

listfields will produce this:

```
joe newman
13509 NE 78th St
Redmond, Wa 98062
```

Note the use of the tr command to transpose colons to linefeeds.

MKFILES

```
#       Usage: mkfiles pref [quantity]
#       Makes "quantity" files, named pref1, pref2, ...
#       Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
        > $1$i
        i="`expr $i + 1`"
done
```

The *mkfiles* procedure uses output redirection to create zero-length files. The expr command is used for counting iterations of the while loop.

NULL

```
#       Usage: null files
#       Create each of the named files as an empty file.
for each file
do
        >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
#       Usage: phone initials ...
#       Prints the phone numbers of the
#       people with the given initials.
echo 'inits     ext         home'
grep "^$1" <<-END
        jfk     1234        999-2345
        lbj     2234        583-2245
        hst     3342        988-1010
        jqa     4567        555-1234
END
```

This procedure is an example of using an in-line input script to maintain a small data base.

TEXTFILE

```
if test "$1" = "-s"
then
#       Return condition code
        shift
        if test -z " `$0 $*`" # check return value
        then
                exit 1
        else
                exit 0
        fi
fi

if test $# -lt 1
then    echo "$0: Usage: $0 [ -s ] file ..." 1>&2
        exit 0
fi

file $* | fgrep 'text' | sed 's/:             .*//'
```

To determine which files in a directory contain only textual information, *textfile* filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr `textfile *` | lpr
```

This procedure also uses an -s flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```
#       Usage: writemail message user
#       If user is logged in,
#       writes message to terminal;
#       otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by $1 is piped to both the write command and, if write fails, to the mail command.

7.16 Shell 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 nonblank characters*
name:	*a sequence of letters, digits, or underscores starting with a letter*
digit:	**0 1 2 3 4 5 6 7 8 9**

Metacharacters and Reserved Words

a. Syntactic

|	Pipe symbol
&&	And-if symbol
||	Or-if symbol
;	Command separator
;;	Case delimiter
&	Background commands
( )	Command grouping
<	Input redirection
<<	Input from a here document
>	Output creation
<	Output append
#	Comment to end of line

b. Patterns

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

c. Substitution

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

d. Quoting

\	Quote next character as literal with no special meaning
'...'	Quote enclosed characters excepting the back quotation marks (`)
"..."	Quote enclosed characters excepting: $ ` \ "

e. Reserved words

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

Chapter 8
BC: A Calculator

8.1 Introduction

BC is a program that can be used as an arbitrary precision arithmetic calculator. BC's output 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. Although you can write substantial programs with BC, it is often used as an interactive tool for performing calculator-like computations. The language supports a complete set of control structures and functions that can be defined and saved for later execution. The syntax of BC has been deliberately selected to agree with the C language; those who are familiar with C will find few surprises. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Common uses for BC are:

— Computation with large integers.

— Computations accurate to many decimal places.

— Conversions of numbers from one base to another base.

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 simply by setting the output base equal to 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine, so manipulation of numbers with many hundreds of digits is possible.

8.2 Demonstration

This demonstration is designed to show you:

— How to get into and out of BC.

— How to perform simple computations.

— How expressions are formed and evaluated.

— How to assign values to registers.

A normal session with BC begins by invoking the program with the command:

```
bc
```

To exit BC type

```
quit
```

or press CNTRL-D. Once you have entered BC, you can use it very much like a normal calculator. As with the XENIX shell, commands are read as command-lines, so each line that you type must be terminated by a RETURN. Throughout this chapter, the RETURN is implied at the end of each command line. Within BC, normal processing of other keys, such as BKSP and INTERRUPT, also works.

For example, type the simple integer 5:

```
5
```

Output is immediately echoed on the next line to the standard output, which is normally the terminal screen:

```
5
```

Here "5" is a simple numeric expression. However, if you type the expression

```
5*5.25
```

(where the star (*) is the multiplication operator) a computation is executed and the result printed on the next line:

```
26.25
```

What has happened here is that the line "5*5.25" has been evaluated, i.e., the expression has been reduced to its most elementary form, which is the number 26.25. The process of evaluation normally involves some type of computation such as multiplication, division, addition, or subtraction. For example, all four of these operations are involved in the following expression:

```
(10*5)+50-(50/2)
```

When this expression is evaluated, the subexpressions within parentheses are evaluated first, just as they would be with simple algebra, so that an intermediate step in the evaluation is "50+50-25" which ultimately reduces to the number "75".

The simple addition

```
10.45+5.5555555
```

produces the output:

```
16.0055555
```

Note how precision is retained in the above result.

The two-part multiplication

```
(8*9)*7
```

produces the answer:

```
504
```

The last part of this demonstration shows you how to store values in special alphabetic registers. For example, type:

```
a=100 ; b=5
```

What happens here is that the registers "a" and "b" are assigned the values 100 and 5, respectively. The semicolon is used here to place multiple BC statements on a single line, just as it is used in the XENIX shell. This command line produces no output because assignment statements are not considered expressions. However, the registers "a" and "b" can now be used in expressions. Thus you can now type

```
a*b; a+b
```

to produce:

```
500
105
```

To exit BC, remember to type

```
quit
```

or press CNTRL-D.

This ends the demonstration. Following sections describe use of BC in more detail. The final section of this chapter is a BC language reference.

8.3 Tasks

This section describes how to perform common BC tasks. Mastery of these tasks should turn you into a competent BC user.

8.3.1 Computing with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type

```
142857 + 285714
```

and press RETURN, BC responds immediately with the line:

```
428571
```

Other operators also can be used. The complete list includes:

```
+  -  *  /  %  ^
```

They indicate addition, subtraction, multiplication, division, modulo (remaindering), and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error message.

Any term in an expression can be prefixed with a minus sign to indicate that it is to be negated (this is the "unary" minus sign). For example, 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 exponentiation (^) performed first, then multiplication (*), division (/), modulo (%), and finally, addition (+), and subtraction (-). The contents of parentheses are evaluated before expressions outside the parentheses. All of the above operations are performed from left to right, except exponentiation, which is performed from right to left. Thus the following 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 convention that a/b*c is equivalent to (a/b)*c.

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way, thus the statement

```
x = x + 3
```

has the effect of increasing by 3 the value of the contents of the register named "x". When, as in this case, the outermost operator is the assignment operator (=), then the assignment is performed but the result is not printed. There are 26 available named storage registers, one for each letter of the alphabet.

There is also a built-in square root function whose result is truncated to an integer (See also Section 8.5, "Scaling"). For example, the lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

8.3.2 Specifying Input and Output Bases

There are special internal quantities in BC, called *ibase* and *obase*. *Ibase* is initially set to 10, and determines the base used for interpreting numbers that are read by BC. For example, the lines

```
ibase = 8
11
```

produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. However, beware of trying to change the input base back to decimal by typing:

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement has no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15, respectively. These characters *must* be uppercase and not lowercase. The statement

```
ibase = A
```

changes you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted; however no mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

Obase is used as the base for output numbers. The value of *obase* is initially set to a decimal 10. The lines

```
obase = 16
1000
```

produce the output line:

```
3E8
```

This is interpreted as a three-digit hexadecimal number. Very large output bases are permitted. For example, large numbers can be output in groups of five digits by setting *obase* to 100000. Even strange output bases, such as negative bases, and 1 and 0, are handled correctly.

Very large numbers are split across lines with seventy characters per line. A split line that continues on the next line ends with a backslash (\). Decimal output conversion is fast, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow.

Remember that *ibase* and *obase* do not affect the course of internal computation or the evaluation of expressions; they only affect input and output conversion.

8.3.3 Scaling Quantities

A special internal quantity called *scale* is used to determine the scale of calculated quantities. Numbers can 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:

Addition, subtraction
: The scale of the result is the larger of the scales of the two operands. There is never any truncation of the result.

Multiplication
: 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*.

Division
: The scale of a quotient is the contents of the internal quantity, *scale*.

Modulo
: The scale of a remainder is the sum of the scales of the quotient and the divisor.

Exponentiation
: The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.

Square Root
: 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 is performed without rounding.

The contents of *scale* must be no greater than 99 and no less than 0. It is initially set to 0.

The internal quantities *scale*, *ibase*, and *base* can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of *scale* by one, and the line

```
scale
```

causes the current value of *scale* to be printed.

The value of *scale* retains its meaning as a number of decimal digits to be retained in internal computation even when *ibase* or *obase* are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

8.3.4 Using Functions

The name of a function is a single lowercase letter. Function names are permitted to use the same letters as 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 returned value of the function is 0; in the second, it is the value of the expression in parentheses.

Variables used in functions 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 can be called recursively and the automatic variables at each call level 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 follows:

```
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.

If the function "a" is defined as shown above, then the line

```
a(7,3.14)
```

would print the result:

```
21.98
```

Similarly, the line

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

would cause the value of "x" to become 60.

Functions can require no arguments, but still perform some useful operation or return a useful result. Such functions are defined and called using parentheses with nothing between them. For example:

```
b ()
```

calls the function named *b*.

8.3.5 Using Subscripted Variables

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable and indicates an array element. The variable name is the name of the array and the expression in brackets is called the

subscript. Only one-dimensional arrays are permitted in BC. 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 can be freely used in expressions, in function calls and in return statements.

An array name can be used as an argument to a function, as in:

```
f(a[ ])
```

Array names can also be declared as automatic in a function definition with the use of empty brackets:

```
define f(a[ ])
auto a[ ]
```

When an array name is so used, the entire contents of the array are copied for the use of the function, then thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other context.

8.3.6 Using Control Statements: if, while and for

The if, while, and for statements are used to alter the flow within programs or to cause iteration. The range of each of these statements is a following statement or compound statement consisting of a collection of statements enclosed in braces. They are written as follows:

```
if ( relation) statement
while ( relation) statement
for ( expression1; relation; expression2)statement

if ( relation) { statements }
while ( relation) { statements }
for ( expression1; relation; expression2) { statements }
```

A relation in one of the control statements is an expression of the form

expression1 rel-op expression2

where the two expressions are related by one of the six relational operators:

```
< > <= >= == !=
```

Note that a double equal sign (==) stands for "equal to" and an exclamation-equal sign (!=) stands for "not equal to". The meaning of the remaining relational operators is their normal arithmetic and logical meaning.

Beware of using a single equal sign (=) instead of the double equal sign (==) in a relational. Both of these symbols are legal, so you will not get a diagnostic message. However, the operation will not perform the intended comparison.

The if statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in the sequence.

The while statement causes repeated execution of its range 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 statement.

The for statement begins by executing *expression1*. Then the relation is tested and, if true, the statements in the range of the for statement are executed. Then *expression2* is executed. The relation is tested, and so on. The typical use of the for statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10.

The following 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.

The following is the definition of a function that computes 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 to 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
        }
}
```

8.3.7 Using Other Language Features

Some language features that every user should know about are listed below.

— Normally, statements are typed one to a line. It is also permissible to type several statements on a line if they are separated by semicolons.

— If an assignment statement is placed in parentheses, it then has a value and 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.

The following is an example of a use of the value of an assignment statement even when it is not placed in parentheses:

```
x = a[i=i+1]
```

This causes a value to be assigned to "x" and also increments "i" before it is used as a subscript.

— The following constructions work in BC in exactly the same manner as they do in the C language:

Construction	Equivalent
x=y=z	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 constructions, if you type one inadvertently, something legal but unexpected may happen. Be aware that 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".

— The comment convention is identical to the C comment convention. Comments begin with "/*" and end with "*/".

— There is a library of math functions that can be obtained by typing

    bc -l

  when you invoke BC. This command loads the library functions sine, cosine, arctangent, natural logarithm, exponential, and Bessel functions of integer order. These are named "s", "c", "a", "l", "e", and "j(n,x)", respectively. This library sets *scale* to 20 by default.

— 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 can load your own programs and function definitions.

8.4 Language Reference

This section is a comprehensive reference to the BC language. It contains a more concise description of the features mentioned in earlier sections.

8.4.1 Tokens

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

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

Identifiers — There are three kinds of identifiers: ordinary identifiers, array identifiers and function identifiers. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, enclosing an optional expression describing a subscript. Arrays are singly dimensioned and can contain up to 2048 elements. Indexing begins at 0 so an array can be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, enclosing optional 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.

Keywords — The following are reserved keywords:

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

Constants — Constants are arbitrarily long numbers with an optional decimal point. The hexadecimal digits A-F are also recognized as digits with decimal values 10–15, respectively.

8.4.2 Expressions

All expressions can be evaluated to a value. The value of an expression is always printed unless the main operator is an assignment. The precedence of expressions (i.e., the order in which they are evaluated) is as follows:

Function calls

Unary operators

Multiplicative operators

Additive operators

Assignment operators

Relational operators

There are several types of expressions:

Named expressions
: Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

identifiers
: Simple identifiers are named expressions. They have an initial value of zero.

array-name[*expression*]
: Array elements are named expressions. They have an initial value of zero.

scale, ibase and obase
: The internal registers *scale*, *ibase*, and *obase* are all named expressions. *Scale* is the number of digits after the decimal point to be retained in arithmetic operations and has an initial value of zero. *Ibase* and *obase* are the input and output number radixes respectively. Both *ibase* and *obase* have initial values of 10.

Constants
: Constants are primitive expressions that evaluate to themselves.

Parenthetic Expressions
: An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter normal operator precedence.

Function Calls
: Function calls are expressions that return values. They are discussed in section 8.10.3.

8.4.3 Function Calls

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. The

syntax is as follows:

function-name ([*expression* [, *expression* ...]])

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 0 if no expression is provided or if there is no return statement. Three built-in functions are listed below:

sqrt(*expr*) The result is the square root of the expression and 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(*expr*) The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale(*expr*) The result is the scale of the expression. The scale of the result is zero.

8.4.4 Unary Operators

The unary operators bind right to left.

-expr The result is the negative of the expression.

++named_expr The named expression is incremented by one. The result is the value of the named expression after incrementing.

--named_expr The named expression is decremented by one. The result is the value of the named expression after decrementing.

named_expr++ The named expression is incremented by one. The result is the value of the named expression before incrementing.

named_expr-- The named expression is decremented by one. The result is the value of the named expression before decrementing.

8.4.5 Multiplicative Operators

The multiplicative operators (*, /, and %) bind from left to right.

*expr*expr* The result is the product of the two expressions. If "a" and "b" are the scales of the two expressions, then the scale of the result is:

min (a+b, max (scale, a, b))

expr/expr The result is the quotient of the two expressions. The scale of the result is the value of *scale*.

expr%expr The modulo 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*.

expr^expr The exponentiation operator binds right to left. 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*b, max (scale, a))

8.4.6 Additive Operators

The additive operators bind left to right.

expr+expr The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expr-expr The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

8.4.7 Assignment Operators

The assignment operators listed below assign values to the named expression on the left side.

named_expr=expr
This expression results in assigning the value of the expression on the right to the named expression on the left.

named_expr=+expr
The result of this expression is equivalent to *named_expr=named_expr+expr*.

named_expr=-expr
The result of this expression is equivalent to *named_expr=named_expr-expr*.

*named_expr=*expr*
The result of this expression is equivalent to

*named_expr=named_expr*expr.*

named_expr=/expr
The result of this expression is equivalent to *named_expr=named_expr/expr.*

named_expr=%expr
The result of this expression is equivalent to *named_expr=named_expr%expr.*

named_expr=^expr
The result of this expression is equivalent to *named_expr=named_expr^expr.*

8.4.8 Relational Operators

Unlike all other operators, the relational operators are only valid as the object of an if or while statement, or inside a for statement. These operators are listed below:

expr<expr

expr>expr

expr<=expr

expr>=expr

expr==expr

expr!=expr

8.4.9 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 to 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. Note that auto arrays are specified by the array namer, followed by empty square brackets.

Automatic variables in BC do not work the same way as in C. 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.

8.4.10 Statements

Statements must be separated by a semicolon or a newline. Except where altered by control statements, execution is sequential. There are four types of statements: expression statements, compound statements, quoted string statements, and built-in statements. Each kind of statement is discussed below:

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.

Compound statements
: Statements can be grouped together and used when one statement is expected by surrounding them with curly braces ({ and }).

Quoted string statements
: For example

  "string"

  prints the string inside the quotation marks.

Built-in statements
: Built-in statements include **auto, break, define, for, if, quit, return,** and **while.**

  The syntax for each built-in statement is given below:

  Auto statement

  The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition. Syntax of the auto statement is:

  auto *identifier* [, *identifier*]

  Break statement

  The **break** statement causes termination of a **for** or **while** statement. Syntax for the break statement is:

break

Define statement

The define statement defines a function; parameters to the function can be ordinary identifiers or array names. Array names must be followed by empty square brackets. The syntax of the define statement is:

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

For statement

The for statement is the same as:

```
first-expression
while(relation) {
        statement
        last-expression
}
```

All three expressions must be present. Syntax of the for statement is:

for (expression; relation;expression) statement

If statement

The statement is executed if the relation is true. The syntax is as follows:

if (relation) statement

Quit statement

The quit statement stops execution of a BC program and returns control to XENIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an if, for, or while statement. Note that entering a CNTRL-D at the keyboard is the same as typing "quit". The syntax of the quit statement is as follows:

quit

Return statement

The return statement terminates a function, pops its auto variables off the stack, and specifies the result of the function. The result of the function is the result of the expression in

parentheses. The first form is equivalent to "return(0)". The syntax of the return statement is as follows:

return(expr)

While statement

The statement is executed while the relation is true. The test occurs before each execution of the statement. The syntax of the while statement is as follows:

while (relation) statement

Chapter 9
Building a Uucp System

9.1 Introduction

The uucp system is a series of programs designed to permit communication between XENIX systems using dial-up communication lines. Uucp provides file transfer and remote command execution through a batch-type operation. Files are created in a spool directory for processing by the uucp daemons. There are three types of files used for the execution of work:

Data files	Contain data for transfer to remote systems
Work files	Contain directions for file transfers between systems
Execution files	Contain directions for XENIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp	This program creates work and gathers data files in the spool directory for the transmission of files.
uux	This program creates work files, execute files and gathers data files for the remote execution of XENIX commands.
uucico	This program executes the work files for data transmission.
uuxqt	This program executes the execution files for XENIX command execution.

The secondary programs are:

uulog	This program updates the log file with new entries and reports on the status of uucp requests.
uuclean	This program removes old files from the spool directory.

This chapter describes the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

For hardwired communications between XENIX systems, use the Micnet network described in the XENIX *Operations Guide*.

9.2 Uucp – System to System File Copy

The *uucp* program is the user's primary interface with the system. The *uucp* program was designed to look like the cp command. The syntax is

uucp [*option*] ... *source* ... *destination*

where *source* and *destination* may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

−d Make directories when necessary for copying the file.

−c Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

−g*letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

−m Send mail on completion of the work.

The following options are used primarily for debugging:

−r Queue the job but do not start *uucico* program.

−s*dir* Use directory *dir* for the spool directory.

−x*num* Use *num* as the level of debugging output.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "?*||". If a source argument has a *system−name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

```
uucp    *.c    usg!/usr/dan
```

will set up the transfer of all files whose names end with *.c* to the */usr/dan* directory on the *usg* machine.

The source and/or destination names may also contain a *user* prefix. This translates to the login directory on the specified system. For names with partial pathnames, the current directory is prepended to the file name. File names with "../" are not permitted.

The command

```
uucp    usg!dan/*.h    dan
```

will set up the transfer of files whose names end with *.h* in dan's login directory on system *usg* to dan's local login directory.

For each source file, the program will check the source and destination filenames and the system−part of each to classify the work into one of five types:

1. Copy source to destination on local system.
2. Receive files from other systems.
3. Send files to a remote systems.
4. Send files from remote systems to another remote system.
5. Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the −r option was specified).

9.2.1 Copying Files to a Local Destination

A cp command is used to do type 1 work. The −d and the −m options are not honored in this case.

9.2.2 Receiving Files from Other Systems

For type 2 work, a one line work file is created for each file requested and put in the spool directory with the following fields, each separated by a blank.

[1] R

[2] The full pathname of the source or a user/pathname. The *user* part will be expanded on the remote system.

[3] The full pathname of the destination file. If the *user* notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A "−" followed by an option list. (Only the −m and −d options will appear in this list.)

9.2.3 Sending Files to Remote Systems

For type 3 work, a work file is created for each dsource file and the source file is copied into a data file in the spool directory. (A −c option on the *uucp* program will prevent the data file from being made. In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

[1] S

[2] The full pathname of the source file.

[3] The full pathname of the destination or user/filename.

[4] The user's login name.

[5] A "−" followed by an option list.

[6] The name of the data file in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

9.2.4 Copying Files Between Systems

For type 4 and 5 work, *uucp* generates a *uucp* command line and sends it to the remote machine; the remote *uucico* executes the command line.

9.3 Uux – System To System Execution

The *uux* command is used to set up the execution of a XENIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

uux [–] [*option*] ... *command–string*

where *command–string* is made up of one or more arguments. All special shell characters such as "<>|^" must be quoted either by quoting the entire command string or quoting the character as a separate argument. Within the command string, the command and file names may contain a *system–name!* prefix. All arguments which do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) The – option is used to indicate that the standard input for the given command should be inherited from the standard input of the *uux* command. The options, essentially for debugging, are:

–r Do not start *uucico* or *uuxqt* after queuing the job

–x*num* Use *num* as the level of debugging output.

The command

pr abc | uux – usg!lpr

will set up the output of "pr abc" as standard input to an lpr command to be executed on system *usg*.

Uux generates an execute file which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command–files will be put on the execution machine and executed by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The execute file will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

U *user system*

where the *user* and *system* are the requester's login name and system.

Required File Line

F *filename real−name*

where the *filename* is the generated name of a file for the execute machine and *real−name* is the last part of the actual file name (contains no path information) Zero or more of these lines may be present in the execute file. The *uuxqt* program will check for the existence of all required files before the command is executed.

Standard Input Line

I *filename*

The standard input is either specified by a "<" in the command−string or inherited from the standard input of the *uux* command if the − option is used. If a standard input is not specified, */dev/null* is used.

Standard Output Line

O *filename system−name*

The standard output is specified by a ">" within the command−string. If a standard output is not specified, */dev/null* is used. (Note that the use of ">>" is not implemented.)

Command Line

C *command* [*arguments*] ...

The *arguments* are those specified in the command string. The standard input and standard output will not appear on this line. All required files will be moved to the execution directory (a subdirectory of the spool directory) and the XENIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell PATH statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

9.4 Uucico − Copy In, Copy Out

The *uucico* program will perform the following major functions:

— Scan the spool directory for work.

— Place a call to a remote system.

— Negotiate a line protocol to be used.

— Execute all requests from both systems.

— Log work requests and work completions.

Uucico may be started by a system daemon, by one of the *uucp*, *uux*, *uuxqt*, or *uucico* programs, by the user (this is usually for testing), or by a remote system. (The uucico program should be specified as the shell field in the */etc/passwd* file for the uucp logins.)

When started by method a daemon, a program, or the user, the program is considered to be in MASTER mode. In this mode, a connection will be made to a remote system. If

started by a remote system. the program is considered to be in SLAVE mode.

The MASTER mode will operate in one of two ways. If no system name is specified (the −s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

−r1 Start the program in MASTER mode. This is used when *uucico* is started by a program or *cron* shell.

−s*sys* Do work only for system *sys*. If −s is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following operions are used primarily for debugging:

−d*dir* Use directory *dir* for the spool directory.

−x*num* Use *num* as the level of debugging output.

The next part of this section will describe the major steps within the *uucico* program.

9.4.1 Scanning For Work

The names of the work related files in the spool directory have format

type . system−name grade number

where *type* may be "C" for copy command file, "D" for data file, "X" for execute file, *system−name* is the remote system, *grade* is a character, and *number* is a four digit, padded sequence number.

The file

C.res45n0031

is a work file for a file transfer between the local machine and the *res45* machine.

The scan for work is done by looking through the spool directory for work files (files with prefix *C.*). A list is made of all systems to be called. *Uucico* will then call each system and process all work files.

9.4.2 Calling a Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is;

[1] System name

[2] Times to call the system (days-of-week and times-of-day)

[3] Device or device type to be used for call

[4] line speed

[5] phone number if field [3] is "ACU" or the device name (same as field [3]) if not

[6] Login information (multiple fields)

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using device type and line speed fields from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy these fields until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the login information n the last field of *L.sys* is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the SLAVE system. The SLAVE sends a message to let the MASTER know it is ready to receive the system identification and conversation sequence number. The response from the MASTER is verified by the SLAVE and if acceptable, protocol selection begins. The SLAVE can also reply with a call-back required message in which case, the current conversation is terminated.

9.4.3 Selecting Line Protocol

The remote system sends a message

Pproto-list

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks the protocol list for a letter corresponding to an available line protocol and returns a use protocol message. The message has the form

Ucode

where *code* is either a one character protocol letter or "N" which means there is no common protocol.

9.4.4 Processing Work

The initial role of MASTER or SLAVE for the work processing is the mode in which each program starts. (The MASTER has been specified by the **-r1** option.) The MASTER program does a work search similar to the one used in the section "Scanning For Work" above.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S Send a file

R Receive a file

C Copy complete

X Execute a *uucp* command

H Hangup

The MASTER will send *R*, *S*, or *X* messages until all work from the spool directory is complete, at which point an *H* message is sent. The SLAVE will reply with the first letter of the request and either the letter "Y" or "N" for yes or no. For example, the message "SY" indicates that it is okay to send a file.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message "CY" will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a "CN" message is sent. (In the case of "CN", the transferred file will be in the spool directory with a name beginning with "TM".) The requests and results are logged on both systems.

The hangup response is determined by the SLAVE program by a work scan of the spool directory. If work for the remote system exists in the SLAVE's spool directory, an "HN" message is sent and the programs switch roles. If no work exists, an "HY" response is sent.

9.4.5 Terminating a Conversation

When a "HY" message is received by the MASTER it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final "OO" message to the other. The original SLAVE program will clean up and terminate. The MASTER will proceed to call other systems and process work as long as possible or terminate if a -s option was specified.

9.5 Uuxqt – Uucp Command Execution

The *uuxqt* program is used to process execute files generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for execute files (prefix *X.*). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The execute file is described in the section "Uux – System to System Copy" above.

The execution is accomplished by executing the shell command

```
sh -c
```

with the command line after appropriate standard input and standard output have been

opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

9.6 Uulog – Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to append these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

−s*sys* Print entries where *sys* is the remote system name

−u*user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

9.7 Uuclean – Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than three days old. These are usually files for work which can not be completed.

The options available are:

−d*dir* The directory to be scanned is *dir*.

−m Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)

−n*hours* Change the aging time from 72 hours to *hours* hours.

−p*pre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)

−x*num* Use *num* as the level of debugging output desired.

9.8 Security

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.

A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call–back for certain login–ids. See the section "Required Files" below in this chapter.

A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.

The *uuxqt* program comes with a list of commands that it will execute. A PATH shell statement is prepended to the command line as specified in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.

The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (The *uucp*, *uucico*, *uux*, and *uuxqt* should be also owned by uucp and have the setuid bit set.)

9.9 Installing a Uucp System

The uucp system provided with the XENIX Software Development System is already configured for operation on your computer. To install the system, you must edit a few files to provide information about your local site. The following sections provide an overview of the files to be edited and the information required.

During execution of the uucp programs, the uucp system uses files from the following three directories:

program (*/usr/lib/uucp*) This is the directory used for the executable system programs and the system files.

spool (*/usr/spool/uucp*) This is the spool directory used during *uucp* execution.

xqtdir (*/usr/spool/uucp/.XQTDIR*) This directory is used during execution of execute files.

The names given in parentheses above are the default values for the directories. The names *lib*, *program*, *xqtdir*, and *spool* will be used in the following text to represent the appropriate directory names.

9.9.1 Modifying the */etc/systemid* File

You must choose a unique site name for each computer to be directly connected to a uucp line and add the site name to the */etc/systemid* file of the corresponding computer by using a XENIX text editor. The */etc/systemid* file can actually contain two names: the uucp site name, which must appear on the first line of the file, and a Micnet machine name, which must appear on the next line. However, you may decide to have both the uucp site name and Micnet machine name to be the same, in which case, only one name is required. For a description of the file, see *systemid*(M) in the XENIX *Reference Manual*.

9.9.2 Creating the Required Files

There are four files which are required for execution, all of which should reside in the *program* directory. To prepare the uucp system for execution, you must add your own site specific information to these files by editing the files with a XENIX text editor. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

line call-unit speed

where *line* is the device for the line (e.g. cul0), *call-unit* is the automatic call unit associated with *line* (e.g. cua0), Hardwired lines have a number "0" in this field, and *speed* is the line speed.

The line

cul0 cua0 300

defines a system which has device "cul0" wired to a call-unit "cua0" for use at 300 baud.

L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

abb dial-seq

where *abb* is the abbreviation, and *dial-seq* is the dial sequence to call that location. The line

py 165-

causes the entry py7777 to be expanded to 165-7777.

USERFILE

This file contains user accessibility information. It specifies

- The files that can be accessed by a normal user of the local machine
- The files that can be accessed from a remote computer
- The login name used by a particular remote computer
- Whether a remote computer should be called back in order to confirm its identity

Each line in the file has the following format

login,sys [c] *pathname* [*pathname*] ...

where *login* is the login name for a user or the remote computer, *sys* is the system name for a remote computer, c is the optional call-back required flag, and *pathname* is a pathname prefix that is acceptable for *user*.

It is assumed that the login name used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique system name which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

When the program is obeying a command stored on the local machine, MASTER mode, the pathnames allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

When the program is responding to a command from a remote machine, SLAVE mode, the pathnames allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a null system name is used.

When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a null system name.

If a line is found that has the appropriate login and remote system names and also contains a "c", the remote machine is called back before any transactions take place.

The line

```
u,m  /usr/xyz
```

allows machine "m" to login with name "u" and request the transfer of files whose names start with "/usr/xyz"."

The line

```
dan,  /usr/dan
```

allows the ordinary user "dan" to issue commands for files whose name starts with "/usr/dan".

The lines

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allow any remote machine to login with name "u", but if its system name is not "m", it can only ask to transfer files whose names start with "/usr/spool".

The lines

```
root,  /
,  /usr
```

allow any user to transfer files beginning with "/usr" but the user with login "root" can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name The name of the remote system.

time — This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800-1730). The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be "Wk" for any week-day or "Any" for any day. The time should be a range of times (e.g. 0800-1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device — This is either "ACU" or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

speed — This is the line speed for the call (e.g. 300).

phone — The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995-9980). For the hardwired devices, this field contains the same string as used for the device field.

login — The login information is given as a series of fields and subfields in the format

expect send [*expect send*] ...

where *expect* is the string expected to be read and *send* is the string to be sent when the expected string is received. The expect field may be made up of subfields of the form

expect [*-send-expect1*] ...

where *send* is sent if the prior *expect* is not successfully read and *expect1* is the next expected string.

There are two special names available to be sent during the login sequence. The string "EOT" sends an EOT character and the string "BREAK" tries to send a BREAK character. (The BREAK character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file is

```
sys Any ACU 300 mh7654 login uucp ssword: word
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

9.10 Maintaining the System

This section indicates some events and files which must be maintained for the uucp system. You may do some maintenance with shell command files, initiating the files with *crontab* entries. Others will require manual modification. Some sample shell files are given toward the end of this section.

9.10.1 SEQF – sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

Use of this feature is not recommend.

9.10.2 TM – temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

TM.*pid.ddd*

where *pid* is a process–id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

uuclean –pTM

removes all *TM* files older than three days.

9.10.3 LOG – log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

uulog

performs the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new *LOG* entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

9.10.4 STST – system status files

These files are created in the spool directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a talking status when to machines are conversing. The form of the file name is

STST.*sys*

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a talking status. In this case, the file must be removed before a conversation is attempted.

9.10.5 LCK – lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

LCK..*str*

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

9.10.6 Creating Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a shell file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

```
program /uulog
program /uucico -r1
```

Note that the **-r1** option is required to start the *uucico* program in MASTER mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST*, and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
program /uuclean -pTM -pC. -pD.
program /uuclean -pST -pLCK -n12
```

can be used. Note the **-n12** option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the **-n** option will use a three day time limit.

9.10.7 Defining Login Entries

One or more logins should be set up for *uucp*. Each of the */etc/passwd* entries should have *program/uucico* as the shell to be executed (where *program* is the directory containing *uucico*). The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the shell argument limits the login to the use of *uucico* only.

9.10.8 Setting File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico*, and *uuxqt* should be owned by the *uucp* login with the setuid bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard shell for the *uucp* logins.

The *L.sys*, *SQFILE*, and *USERFILE* files which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

Chapter 10
The C−Shell

10.1 Introduction

The C-shell program, *csh*, is a command language interpreter for XENIX system users. The C-shell, like the standard XENIX shell *sh*, is an interface between you and the XENIX commands and programs. It translates command lines typed at a terminal into corresponding system actions, gives you access to information, such as your login name, home directory, and mailbox, and lets you construct of shell procedures for automating system tasks.

This chapter explains how to use the C-shell. It also explains the syntax and function of C-shell commands and features, and shows how to use these features to create shell procedures. The C-shell is fully described in *csh*(CP) in the XENIX *Reference Manual*.

10.2 Invoking the C-shell

You can invoke the C-shell from another shell by using the **csh** command. To invoke the C-shell, type:

```
csh
```

at the standard shell's command line. You can also direct the system to invoke the C-shell for you when you log in. If you have given the C-shell as your login shell in your */etc/passwd* file entry, the system automatically starts the shell when you log in.

After the system starts the C-shell, the shell searches your home directory for the command files *.cshrc* and *.login*. If the shell finds the files, it executes the commands contained in them, then displays the C-shell prompt.

The *.cshrc* file typically contains the commands you wish to execute each time you start a C-shell, and the *.login* file contains the commands you wish to execute after logging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C-shell; there is no corresponding XENIX program for this command. **Set** sets the C-shell variable "ignoreeof" which shields the C-shell from logging out if CNTRL-D is hit. Instead of CNTRL-D, the **logout** command is used to log out of the system. By setting the "mail" variable, the C-shell is notified that it is to watch for incoming mail and notify you if new mail arrives.

Next the C-shell variable "time" is set to 15 causing the C-shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the C-shell will remember the last 10 commands typed in its history list, (described later).

Finally, the XENIX *mail* program is invoked.

When the C-shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you log out (by giving the **logout** command) the C–shell prints

```
logout
```

and executes commands from the file *.logout* if it exists in your home directory. After that, the C–shell terminates and XENIX logs you off the system.

10.3 Using Shell Variables

The C–shell maintains a set of variables. For example, in the above discussion, the variables "history" and "time" had the values 10 and 15. Each C–shell variable has as its value an array of zero or more strings. C–shell variables may be assigned values by the **set** command, which has several forms, the most useful of which is:

```
set name=value
```

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

One of the most important variables is "path". This variable contains a list of directory names. When you type a command name at your terminal, the C–shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you typed. The **set** command with no arguments displays the values of all variables currently defined in the C–shell. The following example shows a typical default values:

```
argv    ()
home    /usr/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
```

This output indicates that the variable "path" begins with the current directory indicated by dot (.), then */bin*, and */usr/bin*. Your own local commands may be in the current directory. Normal XENIX commands reside in */bin* and */usr/bin*.

Sometimes a number of locally developed programs reside in the directory */usr/local*. If you want all C–shells that you invoke to have access to these new programs, place the command

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

in the *.cshrc* file in your home directory. Try doing this, then re–executing you **.login** with the command **source .login**. Type

```
set
```

to see that the value assigned to "path" has changed.

You should be aware that when you log in the C–shell examines each directory that you insert into your path and determines which commands are contained there, except for the current directory which the C–shell treats specially, This means that if commands are added to a directory in your search path after you have started the C–

shell, they will not necessarily be found. If you wish to use a command which has been added after you have logged in, you should give the command

rehash

to the C-shell. **Rehash** causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Since the C-shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built in variables are "home" which shows your home directory, and "ignoreeof" which can be set in your *.login* file to tell the C-shell not to exit when it receives an end-of-file from a terminal. The variable "ignoreeof" is one of several variables whose value the C-shell does not care about; the C-shell is only concerned with whether these variables are set or unset. Thus, to set "ignoreeof" you simply type

set ignoreeof

and to unset it type

unset ignoreeof

Some other useful built-in C-shell variables are "noclobber" and "mail". The syntax

>filename

which redirects the standard output of a command just as in the regular shell, overwrites and destroys the previous contents of the named file. In this way, you may accidentally overwrite a file which is valuable. If you prefer that the C-shell not overwrite files in this way you can

set noclobber

in your *.login* file. typing

date > now

causes an error message if the file *now* already exists. You can type

date >! now

if you really want to overwrite the contents of *now*. The ">!" is a special syntax indicating that overwriting or "clobbering" the file is ok. (The space between the exclamation point (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, described below, and have a totally different effect.)

10.4 Using the C-Shell History List

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

The following figure gives a sample session involving typical usage of the history mechanism of the C-shell. Boldface indicates user input:

```
% cat bug.c
main()

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

In this example, we have a very simple C program that has a bug or two in the file *bug.c*, which we cat out on our terminal. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end-of-line. The C-shell echoed the command, as it would have been typed without use of the history mechanism, and then

executed the command. The compilation yielded error diagnostics, so we now edit the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command that started with the letter "c". If there were other commands beginning with the letter "c" executed recently, we could have said "!cc" or even "!cc:p" which prints the last command starting with "cc" without executing it, so that you can check to see whether you really want to execute a given command.

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

We then ran the **size** command to see how large the binary program images we have created were, and then we ran an "ls −l" command with the same argument list, denoting the argument list:

```
!*
```

Finally, we ran the program *bug* to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file *bug.c*. In order to print the listing at a lineprinter we piped the output to **lpr**, but misspelled it as "lpt". To correct this we used a C−shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with

```
!!
```

and sent its output to the lineprinter.

There are other mechanisms available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in *csh* (CP) the XENIX *Reference Manual*.

10.5 Using Aliases

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

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program *mail*. If you place the C−shell command

```
alias mail newmail
```

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

```
mail bill
```

into a call on *newmail*. Suppose you wish the command **ls** to always show sizes of files, that is, to always use the **–s** option. In this case, you can use the **alias** command to do

```
alias ls ls –s
```

or even

```
alias dir ls –s
```

creating a new command named **dir**. If we then type

```
dir ~bill
```

the C–shell translates this to

```
ls –s /usr/bill
```

Note that the tilde (~) is a special C–shell symbol that represents the user's home directory.

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

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

specifies an **ls** command after each **cd** command. We enclosed the entire alias definition in single quotation marks (') to prevent more substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The "\!*" here substitutes the entire argument list to the prealiasing **cd** command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly the following example defines a command that looks up its first argument in the password file.

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

The C–shell currently reads the *.cshrc* file each time it starts up. If you place a large number of aliases there, C–shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15 is reasonable). Too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

10.6 Redirecting Input and Output

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

```
command > & file
```

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

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the lineprinter. The form

```
command >&! file
```

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

Finally, use the form

```
command >> file
```

to append output to the end of an existing file. If "noclobber" is set, then an error results if *file* does not exist, otherwise the C-shell creates *file*. The form

```
command >>! file
```

lets you append to a file even if it does not exist and "noclobber" is set.

10.7 Creating Background and Foreground Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the C-shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the C-shell creates a job. Each of the following lines creates a job:

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

If the ampersand metacharacter (&) is typed at the end of the commands, then the job is started as a background job. This means that the C-shell does not wait for the job to finish, but instead, immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C-shell. Thus

```
du > usage &
```

runs the *du* program, which reports on the disk usage of your working directory, puts the output into the file *usage* and returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program continues executing in the background until it finishes, even though you can type and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard such as the INTERRUPT or QUIT signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

10.8 Using Built-In Commands

This section explains how to use some of the built-in C-shell commands.

The **alias** command described above is used to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. It may also be given one argument, such as to show the current alias for a given string of characters. For example

```
alias ls
```

prints the current alias for the string "ls".

The **history** command displays the contents of the history list. The numbers given with the history events can be used to reference previous events that are difficult to reference contextually. There is also a C-shell variable named "prompt". By placing an exclamation point (!) in its value the C-shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could type:

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

Note that the exclamation mark (!) had to be escaped even within backslashes.

The **logout** command is used to terminate a login C-shell that has "ignoreeof" set.

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

The **repeat** command is used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could type:

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

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

```
setenv TERM adm3a
```

sets the value of the environment variable "TERM" to "adm3a". The program *env* exists to print out the environment. For example, its output might look like this:

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

The **source** command is used to force the current C-shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file that you wish to take effect before the next time you login.

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

```
time cp /etc/rc /usr/bill/rc
```

displays:

```
0.0u 0.1s 0:01 8%
```

Similarly

```
time wc /etc/rc /usr/bill/rc
```

displays:

```
  52   178  1347 /etc/rc
  52   178  1347 /usr/bill/rc
 104   356  2694 total
0.1u 0.1s 0:00 13%
```

This indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01). The word count command **wc** used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the **wc** command used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands are used to remove aliases and variable definitions from the C−shell.

10.9 Creating Command Scripts

It is possible to place commands in files and to cause C−shells to be invoked to read and execute commands from these files, which are called C−shell scripts. This section describes the C−shell features that are useful when creating C−shell scripts.

10.10 Using the argv Variable

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

```
csh script argument ...
```

where *script* is the name of the file containing a group of C−shell commands and *argument* is a sequence of command arguments. The C−shell places these arguments in the variable "argv" and then begins to read commands from *script*. These parameters are then available through the same mechanisms that are used to reference any other C−shell variables.

If you make the file *script* executable by doing

```
chmod 755 script
```

or

```
chmod +x script
```

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

script

If the file does not begin with a number sign (#) then the standard shell */bin/sh* will be used to execute it.

10.11 Substituting Shell Variables

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

```
echo $argv
```

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

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

```
$?name
```

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable "name". To illustrate, examine the following terminal session (input is in boldface):

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

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

```
$argv[1]
```

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

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

would give "c", and

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

would give:

```
a b
```

Other notations useful in C-shell scripts are

```
$n
```

where *n* is an integer This is shorthand for

```
$argv[ n ]
```

the *n*'th parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current C-shell. Since this process number is unique in the system, it is often used in the generation of unique temporary filenames. The form

```
$<
```

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

```
echo -n 'yes or no?'
set a=($<)
```

writes out the prompt

```
yes or no?
```

without a newline and then reads the answer into the variable "a". In this case "$#a" is 0 if either a blank line or CNTRL-D is typed.

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

10.12 Using Expressions

To construct useful C-shell scripts, the C-shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C-shell with the same precedence that they have in C. In particular, the operations "==" and "!=" compare strings and the operators "&&" and "||" implement the logical AND 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]). These operators test whether the string on the left matches the pattern on the right.

The C-shell also allows file enquiries of the form

-? *filename*

where question mark (?) is replaced by a number of single characters. For example, the expression primitive

-e *filename*

tells whether *filename* exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form

{ *command* }

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the "status" variable examined in the next command. Since "$status" is set by every command, its value is always changing.

For the full list of expression components, see **csh**(CP). in the XENIX *Reference Manual*.

10.13 Using the C-Shell: A Sample Script

A sample C-shell script follows that uses the expression mechanism of the C-shell and some of its control structures:

```
#
# 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. The command executes the other commands between the **foreach** and the matching **end**. for each of the values given between parentheses with the named variable "i" which is set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (*i* in this case) has the value at the last iteration.

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

The other control construct is a statement of the form

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

The placement of the keywords in this statement is not flexible due to the current implementation of the C-shell. The following two formats are not acceptable to the C-shell:

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

and

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

The C–shell does have another form of the if statement:

```
if ( expression ) command
```

which can be written

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

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

The more general **if** statements above also admit a sequence of **else–if** pairs followed by a single **else** and an **endif**, for example:

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

else
        commands
endif
```

Another important mechanism used in C–shell scripts is the colon (:) modifier. We can use the modifier **:r** here 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

```
echo $i $i:r $i:e
```

produces

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing ".bar" and the **:e** modifier leaves only the "bar". Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **csh**(CP) entry in the XENIX *Reference Manual*. It is also possible to use the command substitution mechanism to perform modifications on strings to then reenter the C–shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C–shell limits the number of colon modifiers on a "$" substitution to 1. Thus

```
% echo $i $i:h:t
```

produces

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, we note that the number sign character (#) lexically introduces a C–shell comment in C–shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C–shell. This character can be quoted using "'" or "argument word.

10.14 Using Other Control Structures

The C-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
```

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

Finally, the C-shell allows a **goto** statement, with labels looking like they do in C:

```
loop:
        commands
        goto loop
```

10.15 Supplying Input to Commands

Commands run from C-shell scripts receive by default the standard input of the C-shell which is running the script. It allows C-shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

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

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

The notation

```
<< 'EOF'
```

means that the standard input for the ed command is to come from the text in the C-shell script file up to the next line consisting of exactly EOF. The fact that the EOF is enclosed in single quotation marks ('), i.e., it is quoted, causes the C-shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the C-shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form "1,$" in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign ($) with a backslash (\), i.e.:

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

Quoting the EOF terminator is a more reliable way of achieving the same thing.

10.16 Catching Interrupts

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

```
onintr label
```

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

```
exit (1)
```

to exit with status 1.

10.17 Using Other Features

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

One other thing to note is that the C-shell will not execute C-shell scripts that do not begin with the number sign character (#), that is C-shell scripts that do not begin with a comment.

There is also another quotation mechanism using the double quotation mark ("), which allows only some of the expansion mechanisms we have so far discussed to occur on

the quoted string and serves to make this string into a single word as the single quote (') does

10.18 Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on a particular system, */bin/sh*, */bin/nsh*, and */bin/csh*, you could count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v sh$ /etc/passwd
```

Since these commands are very similar we can use **foreach** to simplify them:

```
$ foreach i ('csh$' 'nsh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C-shell prompts for input with "?" when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of filenames or other words. For example, examine the following terminal session:

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

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

The output of a command within back quotation marks (`) is converted by the C-shell to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier :x exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

10.19 Using Braces with Arguments

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

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

expands to

Astr1B Astr2B ... AstrnB

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

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

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

```
chown root /usr/demo/{file1,file2,...}
```

10.20 Substituting Commands

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

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

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

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

to run the editor **vi** supplying as arguments those files whose names end in *.c* which have the string "TRACE" in them. Command expansion also occurs in input redirected with "<<" and within quotation marks ("). Refer to **csh**(CP) in the XENIX *Reference Manual* for more information.

10.21 Special Characters

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

Syntactic metacharacters

; Separates commands to be executed sequentially

| Separates commands in a pipeline

() Brackets expressions and variable values

& Follows commands to be executed without waiting for completion

Filename metacharacters

/ Separates components of a file's pathname

. 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 of characters

~ Used at the beginning of a filename to indicate home directories

{} Used to specify groups of arguments with common parts

Quotation metacharacters

\ Prevents meta-meaning of following single character

' Prevents meta-meaning of a group of characters

" Like ', but allows variable and command expansion

Input/output metacharacters

< Indicates redirected input

> Indicates redirected output

Expansion/Substitution Metacharacters

$ Indicates variable substitution

! Indicates history substitution

: Precedes substitution modifiers

^ Used in special forms of history substitution

` Indicates command substitution

Other Metacharacters

Begins scratch filenames; indicates C-shell comments

- Prefixes option (flag) arguments to commands

Chapter 11
Using The Visual Shell

11.1 What is the Visual Shell?

The Visual Shell **vsh** is a menu-driven XENIX shell. This chapter describes the use and behavior of the vsh. This chapter assumes that the reader is familiar with some general XENIX concepts, specifically the structure of XENIX filesystems and the nature of a XENIX 'command'. No familiarity with any other shell, however, is assumed. If you are a first-time user of the Visual Shell, please completely read the narrative sections of this chapter.

A 'shell' is a program which passes a command to an operating system, and displays the result of running the command. The XENIX shells can also create 'pipelines' for passing the output of one command to another command or 'redirect' the output into a file.

The other XENIX shells available are **sh** and **csh**. These shells are called 'command-line oriented' shells. This means that the user enters commands one line at a time. The sh and csh shells are full computer languages which require study and some programming knowledge to use effectively. These command-line shells are powerful and efficient.

The **vsh** is a 'menu-oriented' shell. In a menu-oriented shell, the user is given the available commands, or some of the available commands. The user can run the command, by selecting from the menu.

The Visual Shell is a good shell for users who may not want to master a programming language right away just to use XENIX or a specfic XENIX application. All Visual Shell users should additionally become familiar with some command-line shell usage.

Users familiar with command-line shells are in for a pleasant surprise if they try the Visual Shell. Experienced users will appreciate the efficiency and versatility of the Visual Shell. The distinction is very much akin to the difference between a line-oriented text editor and a full-screen editor.

A menu shell can be used effectively with very little study. On the other hand, a menu shell can also restrict the user from using the operating system in creative, possibly more efficient ways. The Microsoft Visual Shell strikes a balance in this regard. The Visual Shell is designed to do all of the things that the command-line shells can do.

11.2 Getting Started with the Visual Shell

This section describes how to enter, obtain help about, and leave the visual shell. This section also describes what you will see on the screen while running the visual shell and how the menus work.

Note the following convention for specifying keystrokes. CTRL refers to the CTRL shift key. CTRL-C means pressing the CTRL and 'c' keys at the same time. ALT refers to the ALT shift key. ALT-H means pressing the ALT and 'H' keys at the same time. Note the irrelevance of case in entering Menu Selection characters. For instance, press either 'Q' or 'q' to run the "Quit" command from the main menu.

11.2.1 Entering the Visual Shell

Log in to XENIX If you are not sure how to log in, consult the Operations Guide or have someone knowlegeable about XENIX help you. When you have a shell prompt (typically '$' or '%'), the operating system is waiting for a command. Enter the command:

vsh

and press RETURN.

11.2.2 Getting Help

If at anytime you are not sure what to do, either run the "Help" Menu Selection or press ALT−H. Refer to the reference section of this chapter for information about the Help command.

11.2.3 Leaving the Visual Shell

To exit the Visual Shell select the Quit command from the main menu. The simplest way to do this is to simply press 'q' or 'Q'. In response to the prompt "Type Y to confirm", enter 'y' or 'Y'. If you don't want to exit the Visual Shell yet (perhaps you pressed 'q' by mistake), enter any other character but 'y' or 'Y'. If you have invoked the visual shell from another shell, as described above, you will need to log out from XENIX by entering CTRL−D or 'logout' and pressing RETURN. If the Visual Shell is your default shell, you will automatically be logged out.

11.3 The Visual Shell Screen

11.3.1 Status Line

The bottom line on the screen is called the 'status line'. The status line displays the name of the current working directory, notifies you if you have mail, and gives the date, time and the name of the operating system.

11.3.2 Message Line

The line above the 'status line' is called the 'message line'. The message line displays special output from XENIX commands, such as error reports.

11.3.3 Main Menu

The next section of the screen above the message line is the 'main menu'. The main menu displays a selection of useful XENIX commands.

The currently selected menu command is highlighted on the screen. To select any command, press the SPACE BAR. The next highlighted command is selected. The BACKSPACE key will move to the previous command. Move through the menu until

you have found the command you want. To run the currently selected command, press RETURN.

You may also enter the first letter of a command to select that command. If you enter the first letter of the command, you do not need to press RETURN.

If you enter a letter which does not correspond to a menu selection, the message

Not a valid option

will be displayed. Try another option.

11.3.4 Command Option Menu

When you have selected a command, the main menu will be replaced with a command option menu. The command option menu gives the options available with the specific command. You must fill in the options with appropriate responses.

If you wish to return to the main menu without running the command, press CTRL−C, (cancel). If you want to run the command with the selected options press RETURN.

The following keystrokes allow editing of option responses.

CTRL−I, CTRL−A, or 'tab'	Move to next field in options menu.
CTRL−Y or DEL	Delete character under cursor.
CTRL−L	Move cursor to character to right of current position in current option field.
CTRL−K	Move cursor to character to left of current position in current option field.
CTRL−P	Move cursor to word in current field to right of the current word.
CTRL−O	Move cursor to word in current field to left of the current word.

11.3.5 Program Output

While running a command, commands given and output (unless redirected) will be displayed above the menu and below the view window. The output *scrolls up:* moves from bottom to top. Lines scrolling off the top of the output window disappear.

Visual Shell command lines are listed with each argument preceded by the number in the argument list enclosed in parentheses. The command is named in the output window by the menu command. Hence, if you run the command **/bin/ls** with the argument **−R**, the output window will display the command line as follows:

Run (1) /bin/ls (2) −R

To change the command line format to reflect the actual XENIX command line generated by the Visual Shell, use the Options Output menu command.

11.3.6 View Window

A menu of currently accessible files and directories can be displayed at the top of the

screen in alphabetical order, left to right, top to bottom. Note that this display is the same as that obtained using the view command. This will be referred to as the 'view window' in this chapter. If the directory list is larger than the current window size, you may scroll through using the key commands given below. To reset the window size, use the 'Window' main menu command.

The currently selected item is highlighted in the view window. Use the arrow keys and other key commands given at the end of this section to move the highlight around the window.

If a directory is being listed, subdirectories are shown enclosed in square brackets. To view a subdirectory, press '=' while the directory is highlighted. To return to the previous directory after viewing a subdirectory, press '−'. The parent directory of the current directory is shown as '[..]'. The current directory is shown as '[.]'. Executable files are preceded by an asterisk. The last modification date of the currently selected item is given at the right margin of the last line of the window. The name of the item in view in the current window is given in the upper right−hand corner of the window.

The view window may also display contents of files. Highlight a file, and press '='. You may scroll through the file using the key commands given below. While viewing a file, the highlighted area covers one line.

If you press '=' while an executable file is highlighted, that file will be run.

If the Visual Shell requires a file or directory name, the currently selected View Window item can be automatically entered in the relevent option field by pressing any directional movement key following selection of the command. This method saves keystrokes and reduces the chance of making typing mistakes. On the other hand, if you wish to explicitly enter a file or directory in an option field, type in the name after selecting the command.

Use these keystrokes to select files from the view window:

WINDOW MOTION KEYS

CTRL−Q	Move to start (first item alphabetically) of view window.
CTRL−Z	Move to end (last item alphabetically) of view window.
CTRL−R CTRL−E	Scroll view window up.
CTRL−R CTRL−S	Scroll view window down.
=	View indicated item, either file or directory. If no view window is present, the current working directory is displayed.
−	Return window display to parent directory of currently listed directory. If viewing a file, exit from viewing that file. Last view window is returned to.

DIRECTIONAL MOVEMENT KEYS

ARROW UP or CTRL-E:	Move highlight up in view window.
ARROW DOWN or CTRL-X:	Move highlight down in view window.
ARROW LEFT or CTRL-S:	Move highlight left in view window.
ARROW RIGHT or CTRL-D:	Move highlight right in view window.

Movement beyond the left or right margin will proceed to the next item on the previous or next line unless at the edge of the view window. Movement beyond the top or bottom edge of the current window will scroll the view window up or down if there are more items in that direction in the view window.

Note that there are two ways to move the highlight around. Either use the keypad arrow keys or the cluster of four keys on the far left of the keyboard 'e', 'x', 's', and 'd' shifted with CTRL.

While viewing a file, the directional movement keys for up and left move the highlight up, and the keys for down and right move the highlighted line down.

11.4 Visual Shell Reference

11.4.1 Visual Shell Default Menu

This section describes the default Visual Shell menu commands and options. The menu options are displayed at the bottom of the screen above the status line.

To invoke a command, move the highlight forwards through the main menu using the space bar or the tab key, or backwards using the backspace key. Or simply press the first letter of the command.

Most commands require entering options. Move the cursor to the field using the SPACE BAR, TAB key or BACKSPACE key, and type your response. To edit the options, refer to the key commands listed above in the section in this chapter labelled "Command Option Menu". To select an item from a View Window listing for insertion in a field, refer to the section in this chapter labelled "View Window".

Note that some options have 'switches' with predefined (default) selections. The currently selected switch setting is highlighted. The default is the parenthesized setting. For instance, in the switch:

Recursive: (yes) no

the default is recursive. To change a switch, select the field and press the SPACE BAR or BACKSPACE.

Copy

The Copy command can copy files and directories. To copy a file, select "File" from the options, to copy a directory, select "Directory". A sub-menu will appear. Enter the file or directory you wish copied in the *from:* field. Enter the file or directory you wish copied to the *to:* field. Note that if the item in the *to:* field already exists, it will be overwritten, so be careful.

The Copy Directory sub-menu has a switch "recursive". If this switch is set to yes, all sub-directories and their contents below the specified directory will be copied.

Delete

The Delete command can remove files and directories. In the *DELETE* name: field enter the name of the file or directory you want to remove. Note that once the file or directory is deleted, the contents are gone forever unless you have another copy, so be careful.

Edit

The Edit command invokes the full-screen editor vi. The current directory will be displayed in the output window. Enter in the option field *EDIT filename:* the name of the file you wish to edit using vi.

To learn vi, consult the document "vi: a Screen Editor" in the XENIX *User's Guide*, and the vi(C) manual page in the XENIX *Reference*. A vi reference card is also available.

Help

The Help command (also available by pressing ALT-H at any time), can give on-line help regarding many aspects of Visual Shell use. The view window will display the help file. Use the menu to select the topic you need help with. For instance, move the highlight to 'Keyboard' using the SPACE BAR and press RETURN to view the help file starting at the 'Keyboard' section. The 'Next' and 'Previous' fields in the menu will scroll through the the help file from the present location one screen at a time. Your work will remain undisturbed. To return from Help, press CTRL-C or select the 'Resume' menu option.

Mail

The Mail command enters the XENIX mail system. There are two options: "Send" and "Read" For more information about mail, refer to the section of the XENIX *Users Guide* titled "Mail" or refer to the **mail**(C) manual page.

Name

The Name command renames an existing file or directory. There are two fields, *From:* and *To:*. Enter the name of the file or directory you want to rename in *From:* and the new name in *To:*

11.4.2 Options

The Options Main Menu Selection provides four sub-menus. These sub-menus run commands which typically are used infrequently or which have irrevocable results.

Directory Option

The Directory command has two sub-menus, Make and Usage.

Make Directory Option This command creates a new directory named what you enter in the *name:* field.

Usage Directory Option Counts the number of disk blocks in the directories specified in the *name:* field. The format is the same as the XENIX command **ls -s**. Refer to the manual page **ls**(C).

FileSystem Option

FileSystem has the five sub−menus: Create, FilesCheck, SpaceFree, Mount and Unmount.

Create FileSystem Option Create FileSystem makes a XENIX filesystem. The Create command performs radical system maintenance and may have irrevocable effects. Care is advised when using Create FileSystem.

The functionality is the same as **mkfs**(C). Consult the **mkfs**(C) manual page before running Create FileSystem. Create FileSystem will prompt you for device, block size, gap number and block number. Refer to the XENIX *Operations Guide* chapter on "Using File Systems". The section "Creating a File System" also explains this command.

FilesCheck FileSystem Option FilesCheck checks the consistency of a XENIX filesystem and attempts repair if damage is detected. The FilesCheck command performs radical system maintenance and may have irrevocable effects. Care is advised when using FilesCheck.

The functionality is the same as **fsck**(C). Consult the **fsck**(C) manual page before running FilesCheck. FilesCheck will prompt you for the device to check.

Output Option

The Output Option command has one switch, *commands like: VShell XENIX"* . The default is VShell. IF VShell is set, the **vsh** form of commands given appear in the upward scrolling output window. If XENIX is specified, the XENIX command line which **vsh** generated will be shown instead.

Permissions Option

The Permissions Option command allows changing the access permissions on files and directories. The functionality is the same as the **chmod**(C) command. Consult the **chmod** manual page if you do not understand the concept of XENIX permissions.

In the *name:* field enter the name of the file or directory you wish to alter the permissions on. You may only alter the permissions on files and directories you own. There are four switches, *who:* , *read:* , *write:* , and *execute:* .

The *who:* switch has four settings, *All* , *Me* , *Group* and *Others* . *All* is the default. *All* refers to yourself, those with the same group id as yourself and others. *Me* refers to yourself. *Group* refers and all others with your group id. *Others* refers to those outside your group.

The read, write and execute switches have two settings, yes and no. The default is yes for *Me*, and no for *Group* and *Others* . This grants the given type of permission to those specified in the *who:* switch. No takes away the given type of permission from those specified in the *who:* switch.

11.4.3 Print

The Print command puts a file or files in the queue for your lineprinter. In the *filename:* option field, enter the file or files you want to print.

11.4.4 Quit

The Quit command exits the Visual Shell. The only option is *Enter Y to confirm:*. Enter 'Y' or 'y' if you really want to quit. Any other key cancels the quit.

11.4.5 Run

The Run command executes a program or shell script. The *name:* option takes the name of an executable file. In the *parameters:* option field enter flags to pass to the executable file. The *output:* option can specify a file to redirect output to or another program to send the output to. Enter a vertical bar '|' in the output field to use the pipe menu.

It is also possible to run an executable file by highlighting the name of the file in the View Window and pressing '='.

11.4.6 View

The View command allows you to inspect without altering the contents of files and directories. View is also available at any time for an item highlighted in the View Window by pressing '='. See the section above labelled 'View Window' for the details of using View.

To alter the height and characteristics of the View Window, use the 'Window' menu option. See the section below labelled "Window".

If you have invoked View from the menu, enter the name of the file or directory you wish to view in the *VIEW name:* field, or select from a directory view window.

To return from any View action to the previously displayed View Window, press the minus key '-'.

If you View a non-executable binary file, non-ascii characters are displayed as the character '@'.

11.4.7 Window

The Window command alters the height and redraw characteristics of the Visual Shell View Window.

The

WINDOW redraw: Yes (No)

switch turns on or off redraw of the view window after running a command.

The *height in lines:* field changes the number of lines displayed in the view window. The minimum window height is 1 lines. The default window height is 5 lines. The maximum window height is 15 lines.

11.4.8 Pipes

XENIX allows output from one program to be passed to another program or to be put in a file. This is called 'piping' or 'pipelining'. If the output is placed in a file it is said to be 'redirected'. Piping is supported in the Visual Shell through the pipe menu.

The Pipe menu is invoked by entering a vertical bar '|' character in any option field named *output:*. For instance, the Run main menu and the Pipe menu itself have an *output:* field. The available Pipe menu commands are Count, Get, Head, More, Run, Sort and Tail. Each Pipe menu sub-command also has an *output:* field, which allows construction of pipelines of arbitrary length.

11.4.9 Count

Count counts words, lines and characters in the input pipe. The default is all of the above. There is a switch for each type of item to count. The Count Pipe Menu option corresponds to the XENIX command **wc**. Consult the manual page **wc**(C) for the functionality.

11.4.10 Get

Get looks for patterns in the input pipe. The pattern may be verbatim, or you may specify a "regular expression" to look for. Regular expressions may contain 'wildcard' characters which represent sets of strings. Consult the manual page **grep**(C) for the available wildcard characters.

The first Get switch is *Unmatched (Yes) No*. If you specify Yes (the default), all lines containing the given pattern will be output. If Unmatched is set to off, all lines not containing the given pattern will be output.

The second Get switch is *ignore case:* which suppresses the case while looking for the regular expression. The default is off.

The third Get switch is *line numbers:*, which reports the line in the input stream which the regular expression was matched on. The default is on.

11.4.11 Head

Head prints a specified number of lines of the input stream starting from the first line. The *lines:* field may be set to specify the number of lines at the head of the input stream to print. The default is 5 lines.

The Head Pipe Menu option corresponds to the XENIX command head. Consult the manual page **head**(C) for the functionality.

11.4.12 More

More allows viewing an input stream one screen at a time. The More Pipe Menu option invokes the XENIX command more. Consult the manual page **more**(C) for the functionality.

11.4.13 Run

The Run Pipe Menu option allows the specification of any command not in the Pipe menu. The functionality is the same as the VisualShell Main Menu Option "Run".

11.4.14 Sort

The XENIX sort utility can be invoked through the Sort Pipe menu option. The input stream is sorted.

The first Sort switch is *order:* < >. Select '>', the default, to sort in ascending order. Select '<' to sort in descending order.

The second Sort switch suppresses the case of characters in the sort. The default is off.

The third Sort switch sorts the input stream assuming an initial numeric field in the input stream. If this switch is off, initial numbers will be sorted in ascii order, which means that a line beginning with '10' will be output before the line beginning with '2'. The default is off.

The fourth Sort switch sorts the input stream in dictionary order, rather than ascii order.

The Sort Pipe Menu option corresponds to the XENIX command sort. Consult the manual page sort(C) for the functionality.

11.4.15 Tail

Tail prints a specified number of lines of the input stream up to the end of the stream. The *lines:* field may be set to specify the number of lines to print. The default is 15 lines.

The Tail Pipe Menu option corresponds to the XENIX command tail. Consult the manual page tail(C) for the functionality.

Appendix A
Ed

A.1 Introduction

Ed is a text editor used to create and modify text. The text is normally a document, a program, or data for a program, thus *ed* is a truly general purpose program. Note that the line editor *ex*, available with other XENIX packages is very similar to *ed*, and therefore this chapter can be used as an introduction to *ex* as well as to *ed*.

A.2 Demonstration

This section leads you through a simple session with *ed*, giving you a feel for how it is used and how it works. To begin the demonstration, invoke *ed* by typing:

```
ed
```

This invokes the editor and begins your editing session. An asterisk "*" prompts for commands to be entered. Initially, you are editing a temporary file that you can later copy to any file that you name. This temporary file is called the "editing buffer," because it acts as a buffer between the text you enter and the file that you will eventually write out your changes to. Typically, the first thing you will want to do with an empty buffer is add text to it. For example, after the prompt, type:

```
a
this is line 1
this is line 2
this is line 3
this is line 4
CNTRL-D
```

This "appends" four lines of text to the buffer. To view these lines on your screen, type,

```
1,4p
```

where the "1,4" specifies a line number range and the p command "prints" the specified lines on the screen.

Now type

```
2p
```

to view line number two. Next type just

```
p
```

This prints out the current line on the screen, which happens to be line number

two. By default, most *ed* commands operate on only the current line.

A.3 Basic Concepts

This section illustrates some of the basic concepts that you need to understand to effectively use *ed*.

A.3.1 The Editing Buffer

Each time you invoke *ed*, an area in the memory of the computer is allocated on which you will perform all of your editing operations. This area is called the "editing buffer". When you edit a file, the file is copied into this buffer where you will work on the copy of the original file. Only when you write out your file do you affect the original copy of the file.

A.3.2 Commands

Commands are entered by typing them at your keyboard. Like normal XENIX commands, entry of a command is ended by typing a NEWLINE. After you type NEWLINE the command is carried out. In the following examples, we will presume that entry of each command is completed by typing a NEWLINE, although this will not be explicitly shown in our examples. Most commands are single characters that can be preceded by the specification of a line number or a line number range. By default, most commands operate on the "current line", described below in the section on "Line Numbers". Many commands take filename or string arguments that are used by the command when it is executed.

A.3.3 Line Numbers

Any time you execute a command that changes the number of lines in the editing buffer, *ed* immediately renumbers the lines. At all times, every line in the editing buffer has a line number. Many editing commands will take either single line numbers or line number ranges as prefixing arguments. These arguments will normally specify the actual lines in the editing buffer that are to be affected by the given command. By default, a special line number called "dot" specifies the current line.

A.4 Tasks

This section discusses the tasks you perform in everyday editing. Frequently used and essential tasks are discussed near the beginning of this section. Seldom-used and special-purpose commands are discussed later.

A.4.1 Entering and Exiting The Editor

The simplest way to invoke *ed* is to type:

```
ed
```

The most common way, however, is to type:

```
ed filename
```

where *filename* is the name of a new or existing file.

To exit the editor, all you need to do is type:

```
q
```

If you have not yet written out the changes you have made to your file, *ed* warns you that you will lose these changes by printing the message:

```
?
```

If you still want to quit, type another q. In most cases you will want to exit by typing:

```
w
q
```

so that you first write out your changes and only *then* exit the editor.

A.4.2 Appending Text: a

Suppose that you want to create some text starting from scratch. This section shows you how to put text in a file, just to get started. Later we'll talk about how to change it.

When you first invoke *ed*, it is like working with a blank piece of paper—there is no text or information present. These must be supplied by the person using *ed*, usually by typing in the text, or by reading it in from a file. We will start by typing in some text and discuss how to read files later.

In *ed* terminology, the text being worked on is said to be "kept in a buffer". Think of the buffer as a workspace, or simply as a place where the information that you are going to be editing is kept. In effect, the buffer is the piece of paper on which you will write things, make changes, and finally file away.

You tell *ed* what to do to your text by typing instructions called "commands". Most commands consist of a single letter, each typed on a separate line. *Ed* prompts with an asterisk (*). This prompting can be turned on and off with the

prompt command, P.

The first command we will discuss is append (a) written as the letter "a" on a line by itself. It means "append (or add) text lines to the buffer, as they are typed in." Appending is like writing new material on a piece of paper.

To enter lines of text into the buffer, just type an "a", followed by a RETURN, followed by the lines of text you want, like this:

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

To stop appending, type a line that contains only a period. The period (.) tells *ed* that you have finished appending. (You can also use CNTRL-D, but we will use the period throughout this discussion.) If *ed* seems to be ignoring you, type an extra line with just a period (.) on it. You may find you've added some garbage lines to your text, which you will have to take out later.

After appending is completed, the buffer contains the following three lines:

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

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

To add more text to what you already have, type another a command and continue typing your text.

If you make an error in the commands you type to *ed*, it will tell you by displaying the message:

```
?
error message
```

A.4.3 Writing Out a File: w

You will probably want to save your text for later use. To write out the contents of the buffer into a file, use the write (w) command followed by the name of the file that you want to write to. This copies the contents of the buffer to the specified file, destroying any previous contents of the file. For example, to save the text in a file named *text*, type:

```
w text
```

Leave a space between w and the filename. *Ed* responds by printing the number of characters it has written out. For instance, *ed* might respond with

```
68
```

(Remember that blanks and the newline character at the end of each line are included in the character count.) Writing out a file just makes a copy of the text—the buffer's contents are not disturbed, so you can go on adding text to it. If you invoked *ed* with the command "ed *filename*", then by default a w command by itself will write the buffer out to *filename*.

This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a w command. Writing out the text to a file from time to time as it is being created is a good idea. If the system crashes or if you make some horrible mistake, you will lose all the text in the buffer, but any text that was written out to a file is relatively safe.

A.4.4 Leaving The Editor: q

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

```
q
```

The system responds with the XENIX prompt character. If you try to quit without writing out the file *ed* will print

```
?
```

At that point, write out the text if you want to save it; if not, typing another "q" will get you out of the editor.

Exercise

Enter *ed* and create some text by typing:

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

Write it out by typing:

```
w filename
```

Then leave *ed* by typing:

```
q
```

Next, use the cat command to display the file on your terminal screen to see that everything has worked.

A.4.5 Editing A New File: e

A common way to get text into your editing buffer is to read it in from a file. This is what you do to edit text that you have saved with the w command in a previous session. The edit (e) command places the entire contents of a file in the buffer. If you had saved the three lines "Now is the time", etc., with a w command in an earlier session, the *ed* command

```
e text
```

would place the entire contents of the file *text* into the buffer and respond with

```
68
```

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

If you use the e command to read a file into the buffer, then you don't need to use a filename after a subsequent w command. *Ed* remembers the last filename used in an e command, and w will write to this file. Thus, a good way to operate is this:

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

This way, you can type w from time to time and be secure in the knowledge that if you typed the filename right in the beginning, you are writing out to the proper file each time.

A.4.6 Changing the File to Write Out to: f

You can find out the last file written to at any time using the file (f) command. Just type f without a filename. You can also change the name of the remembered filename with f. Thus a useful sequence is

```
ed precious
f junk
```

which gets a copy of the file named *precious*, then uses f to save the text in the file *junk*. The original file will be preserved as *precious*.

A.4.7 Reading in a File: r

Sometimes you want to read a file into the buffer without destroying what is already there. This function is useful for combining files. This is done with the read (r) command. The command

```
r text
```

reads the file *text* into your editing buffer and adds it to the end of whatever is already in the buffer. For example, pretend that you have performed a read after an edit:

```
e text
r text
```

The buffer now contains *two* copies of *text* (i.e., six lines):

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

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

Exercise

Experiment with the e command by reading and printing various files. You may get the error message

```
?name
cannot open input file
```

where *name* is the name of a nonexistent file. This means that the file doesn't exist, typically because you spelled the filename wrong, or perhaps because you do not have permission to read from or write to that file. Try alternately reading and appending to see how they work. Verify that the command

```
ed file.text
```

is equivalent to

```
ed
e file.text
```

A.4.8 Displaying Lines On The Screen: p

Use the "print"(p) command to print the contents of the editing buffer (or parts of it) on the terminal screen. Specify the lines where you want printing to begin and where you want it to end, separated by a comma and followed by the letter "p". Thus, to print the first two lines of the buffer (that is, lines 1 through 2) type:

```
1,2p
```

Ed responds with:

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

Suppose you want to print *all* the lines in the buffer. You could use "1,3p" as above if you knew there were exactly 3 lines in the buffer. But you will rarely know how many lines there are, so ed provides a shorthand symbol for the line number of the last line in the buffer—the dollar sign ($). Use it this way:

```
1,$p
```

This will print *all* the lines in the buffer (from line 1 to the last line). If you want to stop the printing before it is finished, press the INTERRUPT key. *Ed* then displays

```
?
interrupt
```

and waits for the next command.

To print the *last* line of the buffer, use:

```
$p
```

You can print any single line by typing the line number, followed by a p. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number; there's no need to type the letter p. If you type

```
$
```

ed prints the last line of the buffer.

You can also use $ in combinations like

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

which prints the last two lines of the buffer. This helps when you want to see how far you are in your typing.

The next step is to use address arithmetic to combine the line numbers like dot (.) and dollar sign ($) with plus (+) and minus (-). (Note that "dot" is shorthand for the current line, and is discussed in a later section.) Thus

```
$-1
```

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

```
$-5,$p
```

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

The command

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

prints from three lines before the current line (line dot) to three lines after. The plus (+) can be omitted:

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

is identical in meaning.

Another area in which you can save typing effort in specifying lines is to use plus and minus as line numbers by themselves. For example

```
-
```

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

```
---
```

moves back three lines, as does

```
-3
```

Thus

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

is also identical to

```
?.-3p+3p
```

A.4.9 Displaying The Current Line: dot (.)

Suppose your editing buffer still contains the following six lines:

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

If you type

```
1,3p
```

ed displays

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

Try typing:

```
p
```

This prints

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

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. You can repeat this p command without line numbers, and *ed* will continue to print line 3.

This happens because *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. The line most recently acted on is referred to with a period (.) and is called "dot". Dot is a line number in the same way that dollar ($) is; it means "the current line", or loosely, "the line you most recently did something to". You can use it in several ways. One possibility is to type:

```
.,$p
```

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

Some commands change the value of dot, while others do not. The p command sets dot to the number of the last line printed. In the example above, p sets dot to 6.

Dot is often used in combinations like this one:

```
.+1
```

Or equivalently:

```
.+1p
```

This means "print the next line" and is one way of stepping slowly through the editing buffer. You can also type

```
.-1
```

This means "print the line *before* the current line". This enables you to go backwards through the file if you wish. Another useful command is something like

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

which prints the previous three lines.

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

```
.=
```

Ed responds by printing the value of dot. Essentially, p can be preceded by zero, one, or two line numbers. If no line number is given, *ed* prints the "current line", the line that dot refers to. If one line number is given (with or without the letter p), *ed* prints that line (and dot is set there); and if two line numbers are given, *ed* prints all the lines in that range (and sets dot to the last line printed).

If two line numbers are specified, the first cannot be bigger than the second.

Pressing RETURN once causes printing of the next line. It is equivalent to:

```
.+1p
```

Try it. Next, try typing a minus sign (-) by itself; it is equivalent to typing

```
.-1p
```

Exercise

Create some text using the a command and experiment with the p command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print lines in reverse order using "3,1p" do not work.

A.4.10 Deleting Lines: d

Suppose you want to get rid of the three extra lines in the buffer. Use the delete (d) command. Its action is similar to that of p, except that d deletes lines instead of printing them. The lines to be deleted are specified for d exactly as they are for p. Thus, the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left in our example, as you can check by typing:

```
1,$p
```

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

Exercise

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

Try using line numbers with a, r, and w, as well. You will find that a appends lines *after* the line number that you specify (rather than after dot); that r reads in a file *after* the line number you specify (not necessarily at the end of the buffer); and that w writes out exactly the lines you specify, not the whole buffer. These variations are sometimes useful. For instance, you can insert a file at the

beginning of a buffer by typing

```
0r filename
```

and you can enter lines at the beginning of the buffer by typing:

```
0a
[input text here]
.
```

Notice that typing

```
.w
```

is very different from typing

```
.
w
```

since the former writes out only a single line and the latter writes out the whole file.

A.4.11 Performing Text Substitutions: s

One of the most important *ed* commands is the substitute (s) command. This is the command that is used to change individual words or letters within a line or group of lines. It is the command used to correct spelling mistakes and typing errors.

Suppose that, due to a typing error, line 1 says:

```
Now is th time
```

The letter "e" has been left off of the word "the". You can use s to fix this up as follows:

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

This substitutes for the characters "th", the characters "the", in line 1. To verify that the substitution has worked, type

```
p
```

to get

```
Now is the time
```

which is what you wanted. Notice that dot must be the line where the substitution took place, since the p command printed that line. Dot is always

set this way with the s command.

The syntax for the substitute command follows:

[starting-line,ending-line]s/pattern/replacement/cmds

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. Changing *every* occurrence is discussed later in this section. The rules for line numbers are the same as those for p, except that dot is set to the last line changed. (If no substitution takes place, dot is *not* changed. This causes printing of the error message:

```
?
search string not found
```

Thus, you can type

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

and correct the first spelling mistake on each line in the text.

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

```
s/something/something else/p
```

which makes a correction on the current line, then prints it to make sure the correction worked out right. If it didn't, you can try again. (Notice that the p is on the same line as the s command. With few exceptions, p can follow any command; no other multicommand lines are legal.)

It is also legal to type

```
s/string//
```

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

```
Nowxx is the time
```

you could type

```
s/xx//p
```

to get

```
Now is the time
```

Notice that two adjacent slashes mean "no characters", not a space. There *is* a difference.

Exercise

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

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

This results in:

```
on the other side of the coin
```

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

```
s/ ... / ... /g
```

Try using characters other than slashes to delimit the two sets of characters in the **s** command—anything should work except spaces or tabs.

A.4.12 Searching

Now that you've mastered the substitute command, you can move on to mastering another important concept: context searching.

Suppose you have the original three-line text in the buffer:

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

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

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

```
/string of characters we want to find/
```

For example, the *ed* command

```
/their/
```

is a context search sufficient to find the desired line—it will locate the next occurrence of the characters between the slashes (i.e., "their"). Note that you do not need to type the final slash. The above search command is the same as typing:

```
/their
```

The search command sets dot to the line on which the pattern is found and prints it for verification:

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

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

```
?
search string not found
```

Otherwise, *ed* prints the line it found. You can also search *backwards* in a file for search strings by using question marks instead of slashes. For example

```
?thing?
```

searches backwards in the file for the word "thing" as does

```
?thing
```

This is especially handy when you realize that the string you want is backwards from the current line.

The slash and question mark are the only characters you can use to delimit a context search, though you can use any character in a substitute command. If you get unexpected results using any of the characters

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

read Section A.5, "Context and Regular Expressions".

You can do both the search for the desired line *and* a substitution at the same time, like this:

```
/their/s/their/the/p
```

This yields:

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

The above command contains three separate actions. The first is a context search for the desired line, the second is the substitution, and the third is the printing of the line.

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

Suppose the buffer contains the three familiar lines

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

The *ed* line numbers

```
/Now/+1
/good/
/party/-1
```

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

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. For instance, you could print all three lines by typing

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or any similar combination. The first combination is better if you don't know how many lines are involved.

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

Suppose you search for

```
/horrible thing/
```

and when the line is printed you discover that it isn't the "horrible thing" that you wanted, so it is necessary to repeat the search. You don't have to retype the search, because the construction

```
//
```

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

```
??
```

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

You can also use // as the left side of a substitute command, to mean "the most recent pattern". For example, examine:

```
/horrible thing/
```

Ed prints the line containing "horrible thing".

```
s//good/p
```

This changes "horrible thing" to "good". To go backwards and change "horrible thing" to "good", type:

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

Exercise

Experiment with context searching. Scan through a body of text with several occurrences of the same string of characters using the same context search.

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

Try context searching using *?text?* instead of */text/*. This scans lines in the buffer in reverse order instead of normal order, which is sometimes useful if you go too far while looking for a string of characters. It's an easy way to back up in the file you're editing.

If you get unexpected results with any of the characters

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

read Section A.4, "Context and Regular Expressions".

A.4.13 Changing and Inserting Text: c and i

This section discusses the change (c) command, which is used to change or replace one or more lines, and the insert (i) command, which is used for inserting one or more lines.

The c command is used to replace a number of lines with different lines that you type at the terminal. For example, to change lines ".+1" through "$" to something else, type:

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

The lines you type between the c command and the dot (.) will replace the originally addressed lines. This is useful in replacing a line or several lines that have errors in them.

If only one line is specified in the c command, then only that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of a period to end the input. This works just like the period in the append command and must appear by itself on a new line. If no line number is given, the current line specified by dot is replaced. The value of dot is set to the last line you typed in. Note that the terminating period and the line referenced by dot are completely different: the first is used simply to terminate a command, the second points at a specific line of text.

The i command is similar to the append command. For example

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

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

Exercise

The c command is like a combination of delete followed by insert. Experiment to verify that

```
start,endd
i
[text]
.
```

is almost the same as

```
start,endc
[text]
.
```

These are not *precisely* the same if the last line gets deleted.

Experiment with a and i to see that they are similar, but not the same. Observe that

```
line-numbera
[text]
.
```

appends *after* the given line, while

```
line-numberi
[text]
.
```

inserts *before* it. If no line number is given, i inserts before line dot, while a appends after line dot.

A.4.14 Moving Lines: m

The move (m) command lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer

at the end instead. You *could* do it by typing

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

where *temp* is the name of a temporary file. However, you can do it more easily with the m command:

```
1,3m$
```

This will move lines 1 through 3 to the end of the file.

The general case is

*start-line,end-line*m*after-this-line*

There is a third line to be specified: the place where the moved text gets put. Of course, the lines to be moved can be specified by context searches. If you had

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

you could reverse the two paragraphs like this:

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

Notice the -1. The moved text goes *after* the line mentioned. Dot gets set to the last line moved. Your file will now look like this:

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

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

```
m+
```

moves line dot to one line after the current line dot. If you are positioned on the second line,

```
m--
```

moves line dot to one line after the current line dot.

The m command is more succinct than writing, deleting and rereading. The main difficulty with the m command is that if you use patterns to specify both the lines you are moving and the target, you have to take care to specify them properly, or you may not move the lines you want. The result of a bad m command can be a mess. Doing the job one step at a time makes it easier for you to verify at each step that you accomplished what you wanted. It is also a good idea to issue a w command before doing anything complicated; then if you make a mistake, it's easy to back up to where you were.

For more information on moving text, see Section A.4.18, "Marking Your Spot in a File:k".

A.4.15 Performing Global Commands: g and v

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

For example, the command

```
g/XENIX/p
```

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

For example,

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

prints all the *troff* formatting commands in a file (lines that begin with "."). (For an explanation of the use of the caret (^) and the backslash (\) see Section A.5, "Context and Regular Expressions".

The v command is identical to g, except that it operates on those lines that do *not* contain an occurrence of the pattern. (Mnemonically, the "v" can be thought of as part of the word "inverse".

For example

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

prints all the lines that don't begin with a period (i.e., the actual text lines).

Any command can follow g or v. For example, the following command deletes all lines that begin with ".":

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

This command deletes all empty lines:

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

Probably the most useful command that can follow a global command is the substitute command. For example, we could change the word "Xenix" to "XENIX" every where, and verify that it really worked, with

```
g/Xenix/s//XENIX/gp
```

Notice that we used // in the substitute command to mean "the previous pattern", in this case, "Xenix". The p command executes on each line that matches the pattern, not just on those in which a substitution took place.

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

```
g/^\.P/+
```

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

```
g/topic/?^\.H?p
```

searches for each line that contains the word "topic", scans backwards until it finds a line that begins with a ".H" (a heading) and prints it, thus showing the headings under which "topic" is mentioned. Finally

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

prints all the lines that lie between lines beginning with ".EQ" and ".EN" formatting commands.

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

It is possible to give more than one command under the control of a global command. For example, suppose the task is to change "x" to "y" and "a" to "b" on all lines that contain "thing". Then

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

is sufficient. The backslash (\) signals the **g** command that the set of commands continues on the next line; the **g** command terminates on the first line that does not end with a backslash.

Note that you cannot use a substitute command to insert a new line within a **g** command. Watch out for this.

The command

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

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

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

It is also possible to execute **a**, **c** and **i** commands as part of a global command. As with other multiline constructions, add a backslash at the end of each line except the last. Thus, to add an ".nf" and ".sp" command before each ".EQ" line, type:

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

There is no need for a final line containing a period (.) to terminate the **i** command, unless there are further commands to be executed under the global command.

A.4.16 Displaying Tabs and Control Characters: l

Ed provides two commands for printing the contents of the text you are editing. You should already be familiar with **p**, in combinations like

```
1,$p
```

to print all the lines you are editing, or

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

to change "abc" to "def" on the current line. Less familiar is the "list" (**l**) command which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If

you list a line that contains some of these, l prints each tab as ">" and each backspace as "<". This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The l command also "folds" long lines for printing. Any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash (\), so you can tell it was folded. This is useful for printing lines longer than the width of your terminal screen.

Occasionally, the l command will print a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed, vertical tab, or bell. Each backslash-number combination represents a single ASCII character. Note that numbers are octal and not decimal. When you see such characters, be wary: they may have surprising meanings when printed on some terminals. Often their presence indicates an error in typing, because they are rarely used.

A.4.17 Undoing Commands: u

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

```
u
```

This command does *not* work with the g and v commands.

A.4.18 Marking Your Spot in a File: k

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

```
kx
```

marks the current line with the name "x". If a line number precedes the k, that line is marked. (The mark name must be a single lowercase letter.) You can refer to the marked line with the notation:

```
'x
```

Note the use of the single quotation mark (') here. Marks are very useful for moving things around. Find the first line of the block to be moved and then mark it with:

```
ka
```

Then find the last line and mark it with

```
kb
```

Go to at the place where the text is to be inserted and type:

```
'a,'bm.
```

A line can have only one mark name associated with it at any given time.

A.4.19 Transferring Lines: t

We mentioned earlier the idea of saving lines that are hard to type or used often, to cut down on typing time. Ed provides another command, called t (for transfer) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

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

```
1,$t$
```

duplicates the entire contents that you are editing.

A common use for t is to create a series of lines that differ only slightly. For example, you can type

```
a
Now is the time for all good men to come to the aid of their party.
.
t.                          [make a copy]
s/men/women/                [change it a bit]
t.                          [make third copy]
s/Now is/yesterday was/     [change it a bit]
```

Your file will look like this:

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

A.4.20 Escaping to the Shell: !

Sometimes it is convenient to temporarily escape from the editor to execute a XENIX command without leaving the editor. The shell escape (!) command,

provides a way to do this.

If you type

!command

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

A.5 Context and Regular Expressions

You may have noticed that things don't work right when you use characters such as the period (.), the asterisk (*), and the dollar sign ($) in context searches and with the substitute command. The reason is rather complex, although the solution to the problem is simple. Ed treats these characters as special. For instance, in a context search or the first string of the substitute command, the period (.) means "any character", not a period, so

/x.y/

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

^ . $ [* \ /

The next few subsections discuss how to use these characters to describe patterns of text in search and substitute commands. These patterns are called "regular expressions", and occur in several other important XENIX commands and utilities, including *grep*(C), *sed*(C) (See the XENIX *Reference Manual*).

Recall that a trailing g after a substitute command causes all occurrences to be changed. With

s/this/that/

and

s/this/that/g

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

Either form of the s command can be followed by p or l to print or list the contents of the line. For example, all of the following are legal and mean slightly different things:

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

Make sure you know what the differences are.

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

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

changes the *first* occurrence of "mispell" to "misspell" in each line of the file. But

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

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

If you add a p or l to the end of any of these substitute commands, only the last line changed is printed, not all the lines. We will talk later about how to print all the lines that were modified.

A.5.1 Period: (.)

The first metacharacter that we will discuss is the period (.). On the left side of a substitute command, or in a search, a period stands for *any* single character. Thus the search

```
/x.y/
```

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

```
x+y
x-y
x y
xzy
```

and so on.

Since a period matches a single character, it gives you a way to deal with funny characters printed by l. Suppose you have a line that appears as

```
th\07is
```

when printed with the l command, and that you want to get rid of the \07, which represents an ASCII bell character.

The most obvious solution is to try

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

but this will fail. Another solution is to retype the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big. But for a very long line, retyping is not the best solution. This is where the metacharacter "." comes in handy. Since \07 really represents a single character, if we type

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

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

Since the period matches any single character, the command

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

converts the first character on a line into a comma (,), which very often is not what you intended. The special meaning of the period can be removed by preceding it with a backslash.

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

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

The first period is the line number of the line we are editing, which is called "dot". The second period is a metacharacter that matches any single character on that line. The third period is the only one that really is an honest, literal period. (Remember that a period is also used to terminate input from the a and i commands.) On the *right* side of a substitution, the period (.) is not special. If you apply this command to the line

```
Now is the time.
```

the result is

```
.ow is the time.
```

which is probably not what you intended. To change the period at the end of the sentence to a comma, type

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

The special meaning of the period can be removed by preceding it with a backslash.

A.5.2 Backslash: \

Since a period means "any character", the question naturally arises: what do you do when you really want a period? For example, how do you convert the line

Now is the time.

into

Now is the time?

The backslash (\ ) turns off any special meaning that the next character might have; in particular, "\." converts the "." from a "match anything" into a literal period, so you can use it to replace the period in "Now is the time." like this:

```
s/\./?/
```

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

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

.DE

at the start of a line. The search

```
/.DE/
```

isn't adequate, for it will find lines like

JADE
FADE
MADE

because the "." matches the letter "A" on each of the lines in question. But if you type

```
/\.DE/
```

only lines that contain ".DE" are found.

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

```
/\/
```

won't work, because the backslash (\) isn't a literal backslash, but instead means that the second slash (/) no longer delimits the search. By preceding a backslash with another backslash, you can search for a literal backslash:

```
/\\/
```

You can search for a forward slash (/) with

```
/\//
```

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

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

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

you could use a colon as the delimiter. To delete all the slashes, type

```
s:/::g
```

The result is:

```
exec sys.fort.go etc...
```

When you are adding text with a or i or c, the backslash has no special meaning, and you should only put in one backslash for each one you want.

Exercise

Find two substitute commands, each of which converts the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; you should verify that each works:

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

A.5.3 Dollar Sign: $

The dollar sign "$", stands for "the end of the line". Suppose you have the line

Now is the

and you want to add the word "time" to the end. Use the dollar sign ($) like this:

s/$/ time/

to get

Now is the time

A space is needed before "time" in the substitute command, or you will get:

Now is thetime

You can replace the second comma in the following line with a period without altering the first.

Now is the time, for all good men,

The command needed is:

s/,$/./

to get

Now is the time, for all good men.

The dollar sign ($) here provides context to make specific which comma we mean. Without it the s command would operate on the first comma to produce:

Now is the time. for all good men,

To convert:

Now is the time.

into

Now is the time?

as we did earlier, we can use:

s/.$/?/

Like the period (.), the dollar sign ($) has multiple meanings depending on context. In the following line

```
$s/$/$/
```

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

A.5.4 Caret: ^

The caret (^) stands for the beginning of the line. For example, suppose you are looking for a line that begins with "the". If you simply type

```
/the/
```

you will probably find several lines that contain "the" in the middle before arriving at the one you want. But with

```
/^the/
```

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

The other use of the caret (^) enables you to insert something at the beginning of a line. For example

```
s/^/ /
```

places a space at the beginning of the current line.

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

```
.P
```

you can use the command

```
/^\.P$/
```

A.5.5 Star: *

Suppose you have a line that looks like this:

```
text x     y text
```

where "text" stands for lots of text, and there are an indeterminate number of spaces between the "x" and the "y". Suppose the job is to replace all the spaces between "x" and "y" with a single space. The line is too long to retype, and

there are too many spaces to count.

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

```
s/x *y/x y/
```

The " *" means "as many spaces as possible". Thus "x *y" means an "x", as many spaces as possible, then a "y".

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

```
text x---------y text
```

then all minus signs (-) can be replaced by a single space with the command:

```
s/x-*y/x y/
```

Finally, suppose that the line was:

```
text x.....y text
```

If you blindly type

```
s/x.*y/x y/
```

The result is unpredictable. If there are no other x's or y's on the line, the substitution will work, but not necessarily. The period matches *any* single character so the ".*" matches as many single characters as possible, and unless you are careful, it can remove more of the line than you expected. For example, if the line was like this

```
x text x.....y text y
```

then typing

```
s/x.*y/x y/
```

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

The solution is to turn off the special meaning of the period (.) with the backslash (\):

```
s/x\.*y/x y/
```

Now the substitution works, for "\.*" means "as many periods as possible".

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

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

into

Now is the time.

use ".*" to remove everything after the "for":

```
s/ for.*/./
```

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

```
xy  text  x    y  text
```

where the squares represent spaces, and we said

```
s/x  *y/x  y/
```

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

The way around this is to specify a pattern like

```
/x    *y/
```

which says an "x", a space, then as many more spaces as possible, then a "y", in other words, one or more spaces.

The other pitfall associated with the star (*) again relates to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

abcdef

produces

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this is that

zero is a legitimate number of matches, and there are no x's at the beginning of the line (so that gets converted into a "y"), nor between the "a" and the "b" (so that gets converted into a "y"), and so on. If you don't want zero matches, use

```
s/xx*/y/g
```

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

A.5.6 Brackets: [and]

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

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

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

The construction

```
[0123456789]
```

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

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and there are only three special characters (^,], and -) inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can type:

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

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0-9]; similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase.

Within [ ], the "[" is not special. To get a "]" (or a "-") into a character class, make it the first character.

You can also specify a class that means "none of the following characters". This is done by beginning the class with a caret (^). For example

```
[^0-9]
```

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

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

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

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

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

A.5.7 Ampersand: &

To save typing, the ampersand (&) can be used in substitutions to signify the string of text that was found on the left side of a substitute command. Suppose you have the line

```
Now is the time
```

and you want to make it:

```
Now is the best time
```

You can type:

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

It's unnecessary to repeat the word "the". The ampersand (&) eliminates this repetition. On the *right* side of a substitution, the ampersand means "whatever was just matched", so you can type

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

and the ampersand will stand for "the". This isn't much of a saving if the thing matched is just "the", but if the match is very long, or if it is something like ".*" which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to put parentheses in a line, regardless of its length, type:

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

The ampersand can occur more than once on the right side. For example

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

makes

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

and

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

converts the original line into

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

To get a literal ampersand use the backslash to turn off the special meaning. For example

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

converts the word into the symbol. The ampersand is not special on the left side of a substitute command, only on the right side.

A.5.8 Substituting New Lines

Ed provides a facility for splitting a single line into two or more shorter lines by "substituting in a newline". For example, suppose a line has become unmanageably long because of editing. If it looks like

```
text xy text
```

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

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

This is actually a single command, although it is typed on two lines. Because the backslash (\) turns off special meanings, a backslash at the end of a line makes the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As an example, consider italicizing the word "very" in a long line by splitting "very" onto a separate line, and preceding it with the formatting command ".I". Assume the line in question looks like this:

```
text a very big text
```

The command

```
s/ very /\
.I\
very\
/
```

converts the line into four shorter lines, preceding the word "very" with the line ".I", and eliminating the spaces around the "very" at the same time.

When a new line is substituted in a string, dot is left at the last line created.

A.5.9 Joining Lines

Lines may be joined together, with the j command. Assume that you are given the lines:

```
Now is
 the time
```

Suppose that dot is set to the first line. Then the command

```
j
```

joins them together to produce:

```
Now is the time
```

No blanks are added, which is why a blank was shown at the beginning of the second line.

All by itself, a j command joins the lines signified by dot and dot"+"1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines in a file into one big line and prints it.

A.5.10 Rearranging a Line: \(and \)

Recall that "&" is shorthand for whatever was matched by the left side of an s command. In much the same way, you can capture separate pieces of what was matched. The only difference is that you have to specify on the left side just what pieces you're interested in.

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

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

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

A. B. Smith
C. Jones

It is possible to do this with a series of editing commands, but it is tedious and error-prone.

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

The command

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

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

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

A.6 Speeding Up Editing

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

For example, if you issue a search command like

```
/thing/
```

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

What happens if there is no occurrence of "thing"? Dot is unchanged. This is also true if the cursor was on the only occurrence of "thing" when you issued the command. The same rules hold for searches that use ?...?; the only difference is the direction in which you search.

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

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

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

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

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

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

Experiment to determine what happens if you add *no* lines with an a, c, or i command.

The r command reads a file into the text being edited, at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even type

```
0r
```

to read a file in at the beginning of the text. (You can also type *0a* or *1i* to start adding text at the beginning.)

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

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

which involves a context search.

(Since the w command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you accidentally delete what you're editing.)

The general rule is simple: you are left sitting on the last line changed; if there were no changes, then dot is unchanged. To illustrate, suppose that there are three lines in the buffer, and the line given by dot is the middle one:

```
x1
x2
x3
```

Then the command

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

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

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, only the first line would be changed and printed, and that is where dot would be set.

A.6.1 Semicolon: ;

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

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

Starting at line 1, you would expect the command

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

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

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

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

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

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

```
/thing/
//
```

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

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

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

Closely related is searching for the second to last occurrence of something, as in:

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

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

```
1;/thing/
```

because if "thing" occurs on line 1 it won't be found. The command

```
0;/thing/
```

will work because it starts the search at line 1. This is one of the few places where 0 is a legal line number.

A.6.2 Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you press the INTERRUPT key while ed is executing a command, your file is restored, as much as possible, to what it was before the command began. Naturally, some changes are irrevocable — if you are reading in or writing out a file, making substitutions, or deleting lines. These will be stopped in some unpredictable state in the middle (which is why it usually unwise to stop them). Dot may or may not be changed.

If you are using the print command, dot is not changed until the printing is done. Thus, if you decide to print until you see an interesting line, and then press INTERRUPT, to stop the command, dot will not *not* be set to that line or even near it. Dot is left where it was when the p command was started.

A.7 Cutting and Pasting with the Editor

This section describes how to manipulate pieces of files, individual lines or groups of lines.

A.7.1 Inserting One File Into Another

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

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

and the data contained in *table* has to go there.

To put *table* into the correct place in the file edit *memo*, find "Table 1", and add the file *table* right there:

```
ed memo
/Table 1/
response from ed
.r table
```

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

A.7.2 Writing Out Part of a File

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

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

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

```
/^\.TS/
```

This prints out the found line:

```
.TS
```

Next type

```
.,/^\.TE/w table
```

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

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

The point is that the w command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. If you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it—

don't retype it. For example, in the editor, type:

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

A.8 Editing Scripts

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

For example, suppose you want to change every "Xenix" to "XENIX" and every "USA" to "America" in a large number of files. Put the following lines into the file *script*:

```
g/Xenix/s//XENIX/g
g/USA/s//America/g
w
q
```

Now you can type:

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

This causes ed to take its commands from the prepared file *script*. Notice that the whole job has to be planned in advance, and that by using the XENIX shell command interpreter, you can cycle through a set of files automatically. The dash (-) suppresses unwanted messages from *ed*.

When preparing editing scripts, you may need to place a period as the only character on a line to indicate termination of input from an a or i command. This is difficult to do in ed, because the period you type will terminate input rather than be inserted in the file. Using a backslash to escape the period won't work either. One solution is to create the script using a character such as the at-sign (@) to indicate end of input. Then, later, use the following command to replace the at-sign with a period:

```
s/^@$/./
```

A.9 Summary of Commands

This following is a list of all ed commands. The general form of ed commands is the command name, preceded by one or two optional line numbers and, in the case of e, f, r, and w, followed by a filename. Only one command is allowed per line, but a p command may follow any other command (except e, f, r, w, and q).

a Appends, i.e., adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a period is typed on a new line. The value of dot is set to the last line appended.

c Changes the specified lines to the new text which follows. The new lines are terminated by a period on a new line, as with **a**. If no lines are specified, replace line dot. Dot is set to the last line changed.

d Deletes the lines specified. If none are specified, deletes line dot. Dot is set to the first undeleted line following the deleted lines unless dollar (**$**) is deleted, in which case dot is set to dollar.

e Edits a new file. Any previous contents of the buffer are thrown away, so issue a w command first.

f Prints the remembered filename. If a name follows **f**, then the remembered name is set to it.

g The command g/*string*/*commands* executes *commands* on those lines that contain *string*, which can be any context search expression.

i Inserts lines before specified line (or dot) until a single period is typed on a new line. Dot is set to the last line inserted.

l Lists lines, making visible nonprinting ASCII characters and tabs. Otherwise similar to p.

m Moves lines specified to after the line named after **m**. Dot is set to the last line moved.

p Prints specified lines. If none are specified, print the line specified by dot. A single line number is equivalent to a *line-number*p command. A single RETURN prints ".+1", the next line.

q Quits ed. Your work is not saved unless you first give a w command. Give it twice in a row to abort edit.

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

s The command "s/*string1*/*string2*/" substitutes the pattern matched by *string1* with the string specified by *string2* in the specified lines. If no lines are specified, the substitution takes place only on the line specified by dot. Dot is set to the last line in which a substitution took place, which means that if no substitution takes place, dot remains unchanged. The s command changes only the first occurrence of *string1* on a line; to change multiple occurrences on a line, type a g after the final slash.

t Transfers specified lines to the line named after t. Dot is set to the last line moved.

v The command *v/string/commands* executes *commands* on those lines that *do not* contain *string*.

u Undoes the last substitute command.

w Writes out the editing buffer to a file. Dot remains unchanged.

.= Prints value of dot. (An equal sign by itself prints the value of $.)

!*command*
The line *!cmd-line* causes *cmd-line* to be executed as a XENIX command.

/string/
Context search. Searches for next line which contains this string of characters and prints it. Dot is set to the line where string was found. The search starts at .+1, wraps around from $ to 1, and continues to dot, if necessary.

?*string*?
Context search in reverse direction. Starts search at .–1 , scans to 1, wraps around to $.

Index