PLEXUS

Sys5 UNIX User Guide

98-05083.1 Ver. A July 1, 1985

Sys5 UNIX User Guide

98-05083.1 Ver. A July 1, 1985

PLEXUS COMPUTERS, INC.
3833 North First Street
San Jose, CA 95134
408/943-9433

CONTENTS

Date: MMDDHHMMYY
month day hour minute year

1. INTRODUCTION

The *Sys5 UNIX* User Guide* covers the following topics:

- A description of the features in the UNIX operating system
- A general overview of the capabilities of the UNIX operating system
- Instructions on how to use the UNIX operating system.

Not all of the capabilities of the UNIX operating system are described or illustrated herein, but enough are described so that a new user can become familiar with the use of the UNIX operating system.

Using the available information, users who have more interest than the novice can utilize the information herein to accomplish their tasks with some experimenting and self-teaching.

Throughout this volume, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX Administrator Reference Manual.* Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX User Reference Manual.* Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX Programmer Reference Manual.*

1.1 CAVEATS

Document processing features described throughout the following sections may not be available on your system.

* UNIX is a trademark of Bell Laboratories, Inc.

2. PRIMER

This section of the *Sys5 UNIX User Guide* provides the information that users will need to access the UNIX operating system. It is not intended to be a detailed description. Many of the subjects described are discussed in detail in other sections of this volume or the *Sys5 UNIX User Reference Manual.*

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX Administrator Reference Manual.* Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX User Reference Manual.* Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX Programmer Reference Manual.*

In this primer, software programs that can be executed by users are referred to as **programs**. A program that is in some state of execution is referred to as a **process**. The request typed by the user is referred to as a **command** or "**command line.**"

In this primer, the following graphic conventions are used in examples:

(RETURN)	Indicates that the user should press the RETURN key on the terminal keyboard.
(DEL)	Indicates that the user should press the key marked DEL, DELETE, or RUBOUT (whichever is appropriate for the terminal being used).

2.1 HUMAN INTERFACE

2.1.1 Concept of a Login

The UNIX operating system is accessed by the use of a *login*. A login is used by the system to uniquely identify users. Before the user can access the system, the user must be assigned a login by the system administrator. Every login consists of the following components:

login name

user identification number (uid)

group identification number (gid)

password.

A *login name* is a unique string of letters (should be all lowercase) and/or numbers that identifies an individual to the system. The login name must begin with a letter. In many cases, a person's login name is their real first name, last name, initials, or nickname. Any string of letters and/or digits can be used as your login name, as long as it is *unique* (i.e., different from all other login names). Only the first eight characters of a login name are used by the system. Login names are assigned by the system administrator.

The *uid* of a login is a unique number assigned to each login by the system administrator. This number is used by the system to identify the owners of information stored on the system and the commands that users are executing.

The *gid* is a unique number assigned by the system administrator to each group. This number identifies *groups* of users that have something in common. For example, all logins used by people in the same department (or working on the same project) may have the same gid. The gid is important for security and accounting reasons. The impact of gid numbers on the user and the group that the user belongs to is described later.

The *password* is a string of 13 characters chosen from a 64-character alphabet (., \, 0-9, A-Z, a-z) that serves to control access to a login. The password for a login is the main security feature of the UNIX operating system. Usually, every login is assigned a password. When a user *logs in* to the system, the password (if any) assigned to the login being used is requested. Access to the system is not permitted until the correct password is entered. The user can change a password as needed to ensure that others are not accessing the user's login (and consequently the user's data). Any string of letters, numbers, etc., can be used as a password as long as it is from six to thirteen characters in length and composed of uppercase letters, lowercase letters, numbers, or punctuation.

It is recommended that obvious strings such as the user's social security number, birth date, or other data that could be well known about the user not be used as passwords. If the password is something that is well known about the user, someone could gain access to the user's login with little effort. The more unusual your password, the more effective your security.

2.1.2 Logging In

In order to log in, the power to the terminal must be turned on and the appropriate switches set. Depending on the type of terminal and communication link, the user may need to press the return or break key a couple of times. This is to synchronize your terminal with the system. When communication is established, the system will prompt with:

```
login:
```

The user should type in his/her login name followed by a return. After the system digests your login name, it will prompt for your password with:

Password:

The user should then type his/her password followed by a return. The system does not echo your password on the terminal as you type it in. This is an extra security measure. If you entered your login name and password correctly, the system may print one or more "messages of the day". Following the messages, the system will prompt you with the primary prompt string, which is usually the *$* symbol. If a mistake is made while logging in or the system adminstrator has not set up the user's login on the system, the following error message is printed:

login incorrect

This error message is followed by the login: message. The user should attempt to log in again.

The UNIX operating system has a hierarchy of *directories*. When the system administrator gave the user a login name, the administrator also created a "directory" for the user. This directory ordinarily is the same name as the user login name and is known as the *login* or *home* directory of the user. When the user logs in, the *home* directory becomes the current directory or working directory of the user. Any file created under the login name (assuming no other subdirectories have been created yet) is by default in the home directory. The user may, however, create one or more directories under the *home* directory. The user may then change to subdirectories by appropriate use of a "change directory" command. See **cd**(1) for details. Under a directory or a subdirectory, the user may create files as necessary. The user is the owner of the *home* directory and all subdirectories created under the *home* directory. As the owner, the user has full permission to create, alter, and remove (destroy) all files and subdirectories of the *home* directory. To change from one directory to another, the command **cd** is used.

2.1.3 Logging Off

After completing your work, it is best to log off the system. Before logging off, you should have received the prompt string "$ " from the system. That is, all your commands have been completed; and the system is ready for another command.

The preferred method for logging off is accomplished by typing an American Standard Code for Information Interchange (ASCII) End Of Text (EOT) character which is sometimes called the End-Of-File (EOF). On most terminals, the EOT character is generated by holding down the "CONTROL" key and pressing the lowercase "d" key once. This is also referred to as a *CONTROL-d*. Regardless of the terminal type, the power to it should be turned off when the terminal is no longer needed. For terminal connected via a phone line, you should hang up the phone.

2.1.4 Entering Commands

The UNIX operating system **shell** (command interpreter) serves as the interface between the user and the system. The **shell** accepts requests from the user in the form of a *command line* and invokes the appropriate program to fulfill the request. The **shell** prompts (i.e., notifies) the user when it is ready to accept another request. The prompt of the UNIX operating system **shell** is the primary prompt string which is by default "**$** " (a dollar sign followed by a space).

2.1.4.1 Command Line Syntax

Commands or requests to the **shell** are usually in the form of a single line, that is, a string of one or more words followed by a return. This single line request entered following the prompt is referred to as a "command line". The command line is divided into two major parts—the program name and arguments.

The first word of the command line is the name of the program to be executed. This is referred to as the **command**. All subsequent words are *arguments* to the command. Arguments are used to provide information required by the program.

Spaces and tabs serve as the delimiters for words on the command line. That is, all characters on the command line up to the first space or tab are interpreted as the command. All characters between the first space (or tab) and the second space (or tab) is the first argument, etc. Thus, the syntax for the command line is:

```
command argument argument argument ... ...(RETURN)
```

When spaces or tabs are needed within a single argument, that argument is enclosed by *double* quote marks. For example, to execute a program that requires two arguments such as *john I* and *doe*. The first argument should be *john* and the initial *I*, that is, "john I". The second argument should be *doe*. The required command line in this case would be:

```
command "john I" doe(RETURN)
```

2.1.4.2 Correction and Deletion

All users are likely to make mistakes, especially when typing. The UNIX operating system provides two features to correct command lines. These features are *only* effective for the current line (i.e., you have not ended the line with a return yet).

The first correction feature is the erase character (by default, #), and the second correction feature is the kill character (by default, @). The erase character erases the character preceding it. For example, a command line entered as

```
caf#t the fik#le (RETURN)
```

actually is "cat the file". The first # erases the first f and the second # erases the k. The erase character can be used to erase a series of characters such as in

```
this####the cat had kittens (RETURN)
```

which results in "the cat had kittens". The entire word "this" is erased by the series of # characters following it. The first # erases the s, the second # erases the i, the third # erases the h, and the fourth # erases the t. If you miscount the number of erase characters you need as in

```
this ###the cat had kittens (RETURN)
```

the result would be "ththe cat had kittens". The three erase characters erase the space, the s, and the i preceding them.

If the user needs to enter a # in the command line for some reason, preceding the # with the backslash character (\) will turn off the "erase last character" meaning of the #. For example, a command line entered as

```
thsi##is is the \#7# 7 cat (RETURN)
```

is actually "this is the # 7 cat".

The second correction feature is the kill character. The kill character deletes the entire current line. For example, the user enters the command line

```
command####omma####mmad argm##gmu##ment
```

when the user was trying to enter "command argument". This command line is so full of mistakes and corrections it is hard to determine if it is right. It would be best to delete the entire line and start over. The user can delete the line by ending it with an @ instead of a return. For example in this sequence

```
kat###catteh##he file######## the flie##e@
cat the file (RETURN)
```

the first line is deleted by the @ character. It is much easier to delete it and reenter it (as in the second line of the example).

If the @ character is needed in a line, the backslash character (\) should precede it. For example, entering the line

```
The kill character is a \@ (RETURN)
```

results in "The kill character is a @".

2.1.4.3 Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice (terminal may be in the half-duplex mode) or the RETURN may not cause a line feed or a return to the left margin. The user can often change this by logging out and logging back in. If logging back in fails to correct the problem, check the following areas:

keyboard	Keys such as caps lock, local, block, etc. should not be in depressed position.
dataphone	For terminals connected via phone lines, the baud rate could be incorrect.
switches	The rear panel of your terminal normally has several switches used to control terminal operations. These switches should be set to be compatible with the UNIX operating system.

If all else fails, the description of the **stty**(1) command can be read to determine the appropriate action to take. To get intelligent treatment of tab characters (which are much used in the UNIX operating system) if your terminal does not have tabs, type the command

```
stty −tabs
```

and the system will convert each tab into sufficient blanks to space to the next 8-character field. If your terminal does have hardware tabs, the command **tabs**(1) will set the stops correctly for you.

2.1.4.4 Read—ahead

The UNIX operating system has full read-ahead, which means that the user can type as fast as desired, whenever the user wants, even when some command is already outputting on the terminal. If typing is done during output, the input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So the user can type several commands one after another without waiting for the first to finish or even begin.

2.1.5 Stopping a Program

Most programs can be stopped by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, "DEL" stops whatever the program is doing but leaves you in that program. Hanging up the phone with the talk button depressed will also stop most programs.

2.1.6 Mail

After logging in, the user may sometimes get the following message:

You have mail.

The UNIX operating system provides a postal system so you can communicate with other users of the system. To read your mail, type the following command:

```
mail
```

Your mail will be printed, one message at a time, most recent message first. After each message, **mail**(1) waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the *Sys5 UNIX User Reference Manual.*

How is mail sent to someone else? Assume that "jones" is someone's login name which is recognized by **login**(1). The easiest way to send mail to "jones" is as follows:

mail jones
now type in the text of the letter
on as many lines as you like...
After the last line of the letter
type the character "CONTROL-d",
that is, hold down "CONTROL" and type
a letter "d".

The "CONTROL-d" sequence, often called End -Of-File (EOF), is used throughout the system to mark the end of input from a terminal.

For practice, send mail to yourself. (This is not as strange as it might sound—mail to oneself is a handy reminder mechanism.)

There are other ways to send mail—you can send a previously prepared letter, and you can mail to a number of people all at once. For more details, see **mail**(1).

2.1.7 Writing to Other Users

At some point, out of the blue will come a message like

```
Message from jones tty07...
```

which is accompanied by a startling beep on terminals that have the capability to beep. It means that Jones (jones) wants to talk to you, but unless you take explicit action, you will not be able to talk back. To respond, type the following command:

```
write jones
```

This establishes a 2-way communication path. Now whatever jones types on his terminal will appear on yours and vice versa. However, if you are in the middle of some program, you must get back to a state where you are talking to the command interpreter. Normally, whatever program you are running has to terminate or be terminated. If you are editing, you can escape temporarily from the editor—read the "Tutorial—Text Editor" section of this document. If you are printing and do not want this message in your printout or you simply do not want to be disturbed, enter the following:

```
mesg n
```

If you never wish to be disturbed, add the "**mesg n**" command line to your *.profile*.

A protocol is needed to keep what you type from getting garbled up with what jones types. Typically, a sequence like the following is used:

Jones types "write smith" and waits.

Smith types "write jones" and waits.

Jones now types a message
(as many lines as necessary).
When he is ready for a reply, he
signals it by typing
(o)
which stands for "over".

Now Smith types a reply, also
terminated by
(o).

This cycle repeats until
someone gets tired; he then
signals his intent to quit with
(oo)
for "over and out".

To terminate
the conversation, each side must
type a "CONTROL-d" character alone
at the beginning of a line. ("DELETE" also works.)
When the other person types "CONTROL-d",
you will get the message
EOF
on your terminal.

If you write to someone who is not logged in or who does not want to be disturbed, you will be told. If the target is logged in but does not answer after a decent interval, simply type "CONTROL-d".

2.1.8 On-line Manual

The *Sys5 UNIX User Reference Manual*, *Sys5 UNIX Programmer Reference Manual*, and *Sys5 UNIX Administrator Reference Manual* are kept on-line. If you get stuck on something and can not find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "man command-name". Thus to read up on the **who**(1) command, type

man who

and, of course,

man man

tells all about the **man**(1) command.

3. BASICS FOR BEGINNERS

3.1 Creating Files—The Editor

If you have to type a paper, a letter, or a program, how do you get the information into the machine? These tasks can be performed using the UNIX operating system "text editor". See **ed**(1) and the "TUTORIAL—TEXT EDITOR" section of this volume for a detailed description.

Throughout this section, each reference of the form **name**(N) refers to entries in the *Sys5 UNIX Administrator Reference Manual*, *Sys5 UNIX User Reference Manual*, or *Sys5 UNIX Programmer Reference Manual*, where **name** is the name of the file, and **(X)** is the chapter of the book it is in.

The UNIX operating system "text editor" operates on a "file". Simply stated, a file is just a collection of information stored in the machine. The following text will describe how to make some *files*. To create a file called *junk* with text in it, do the following:

ed junk (invokes the text editor)
a (command to "**ed**" to add text)
now type in
whatever text you want ...
. (signals the end of text addition)

The "**.**" that signals the end of adding text must be at the beginning of a line by itself. Do not forget it, for until it is typed, no other **ed** commands will be recognized—everything you type will be treated as text to be added. Also note that no system prompt appears while you are appending, inserting, or changing text while in the text editor.

After a file exists, the user can do various editing operations on the text which was typed in, such as correcting spelling mistakes, rearranging paragraphs, etc.

Finally, the user must write the information typed into a file with the editor command:

w

The **ed** will respond with the number of characters it wrote into the file *junk*.

Nothing is stored permanently in the *junk* file until the **w** command is used. If the user is editing a file and hangs up before using the **w** command, the changes are not stored in the working file. The data in this case is saved in a file called *ed.hup* which the user can continue working with at the next editing session. But after **w** the information is there permanently. The user can reaccess it any time by typing the following:

```
ed junk
```

Type a **q** command to quit the editor. (If you try to quit without writing, the text editor will print a "**?**" to remind you. A second **q** gets the user out of the text editor regardless.) Now create a second file called *temp* in the same manner. You should now have two files, *junk* and *temp*.

3.2 What Files Are Out There?

The **ls**(1) command lists the names (not contents) of any of the files that the UNIX operating system knows about. If you type

```
ls
```

the response will be

```
junk
temp
```

which are indeed the two files just created.

The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

```
ls -t
```

causes the files to be listed in the order in which they were last changed, most recent first. The **-l** option gives a "long" listing and is used as follows

```
ls -l
```

to produce something like

```
-rw-rw-rw- 1 bwk  bsk 41 Jul 22 02:56 junk
-rw-rw-rw- 1 bwk  bsk 78 Jul 22 12:57 temp
```

The date and time is the date and time of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). The "bwk" is the owner of the file, i.e., the person who created it. The "bsk" identifies the group associated with "bwk". The "-rw–rw–rw–" determines who has permission to read, write, or execute the file. In this case the owner, group, and others all have permission to read (r) and write (w). Note that there is no permission for anyone to execute (x). The first character in "-rw–rw–rw–" is a "-" which indicates this is a file of data. A "d" in the first character would indicate a directory. The remaining nine characters are divided into three sets of permissions. Each set consists of three characters. The three sets correspond to the permissions of the owner, group, and all other users.

Options can be combined: **ls −lt** gives the same thing as **ls −l** but sorted into time order. The user can also name the files interested in, and **ls** will list the information about them only. More details can be found in **ls**(1).

The use of optional arguments that begin with a minus sign (like *−t* and *−lt*) is a common convention for UNIX system programs. In general, if a program accepts such optional arguments, they precede any file name arguments. It is also vital that you separate the various arguments with spaces: **ls−l** is not the same as **ls −l** since the command **ls** must be separated from its argument *−l* by a space.

3.3 Printing Files

Now that you've created a file of text, how can the file be printed so people can look at it? There are several ways to print a file. One simple way to obtain a print is to use the editor, since printing is often done just before making changes anyway. The editor is used to print as follows:

```
ed junk
1,$p
```

The **ed** will reply with the count of the characters in *junk* and then print all the lines in the file. The user can also be selective about the parts of a file to be printed as follows:

```
ed junk
20,35p
```

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time; and sometimes you want to print several, one after the other. So here are a couple of alternatives.

The simplest of all the printing programs is **cat**(1). The **cat** command simply prints on the terminal the contents of all the files named and in the order listed. Thus the files are concatenated and printed. For example:

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two files. The files are simply concatenated onto the terminal.

The **pr**(1) command produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number, and file name at the top of each page, and extra lines to skip over the fold in the paper.

Thus,

```
pr junk temp
```

will print *junk* neatly, then skip to the top of a new page and print *temp* neatly.

The **pr** command can also produce multicolumn output. Inputting

```
pr −3 junk
```

prints *junk* in 3-column format. You can use any reasonable number in place of "3", and **pr** will do its best. The **pr** command has other capabilities also. See **pr**(1) for more information.

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a hard copy printer. See **lp**(1) for more information.

3.4 Moving Files Around

The user is ready for bigger things after gaining experience in creating and printing files. For example, the user can move a file from one place to another (which amounts to giving it a new file name), like this:

```
mv junk precious
```

This means that what used to be named *junk* is now named *precious*. An **ls**(1) command would now result in the following:

```
precious
temp
```

The contents of *junk* are now in *precious*. Notice that the *junk* file no longer exists. Beware that if you move a file to another one that already exists, the already existing file contents are lost *forever*.

If you want to make a copy of a file (i.e., to have two versions of something), use the **cp**(1) command as follows:

```
cp precious temp1
```

This makes a duplicate copy of *precious* in *temp1*.

When you are finished creating and moving files, the files can be removed from the file system by the **rm**(1) command. The command is used as follows:

```
rm temp temp1
```

This will remove both the *temp* and *temp1* files.

The user will get a warning message if one of the named files is not there, but otherwise **rm** like most UNIX system commands does its work silently. There is no prompting or response, and error messages are just occasionally shortened. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

3.5 What's in a File Name

So far we have used file names without ever saying what is a legal name, so it is time for a couple of rules. First, file names are limited to 14 characters, which is enough to be descriptive. Second, although any character can be used in a file name, common sense dictates sticking to ones that are visible and avoiding characters that could be used with other meanings. We have already seen, for example, that in the **ls**(1) command, **ls −t** means to list in time order. So if a file existed whose name was *−t*, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, use only letters, numbers, and the period until you are familiar with the system.

On to some more positive suggestions. Suppose you are typing a large document like a book. Logically, this divides into many small pieces, like chapters and perhaps sections. Physically, it must be divided too, for **ed** will not handle really big (over 90,000 characters) files. Thus the document should be typed as a number of files. One possible method is to have a separate file for each chapter as follows:

```
chap1
chap2
etc. ...
```

Another method is breaking each chapter into several files as follows:

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

It can now be determined at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX system user. To print the whole book, the user could enter the following:

```
pr chap1.1 chap1.2 chap1.3 ...
```

Using the **pr**(1) command like this would be tiring and possibly lead to making mistakes. Fortunately, there is a shortcut. The user can enter:

```
pr chap*
```

The * means "anything at all", so this translates into "print all files whose names begin with *chap* listed in alphabetical order".

This shorthand notation is not a property of the **pr** command by the way. It is system-wide, a service of the program that interprets commands—the "**shell**", **sh**(1). The files in the book can be listed by using

```
ls chap*
```

which produces the following:

```
chap1.1
chap1.2
chap1.3
...
```

The * is not limited to the last position in a file name. The * can be used anywhere and can occur several times. Thus entering

```
rm *junk* *temp*
```

removes all files that contain *junk* or *temp* as any part of their name. As a special case, * by itself matches every file name, so

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

removes *all files*. (Before using the **rm** * command, make sure all files are not needed!)

The * is not the only pattern-matching feature available. To print only chapters 1 through 4 and 9, use the following command:

```
pr chap[12349]*
```

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated as follows:

```
pr chap[1-49]*
```

Letters can also be used within brackets. The **[a-z]** pattern-matching feature matches any character in the range *a* through *z*.

The **?** pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls −l chap?.1
```

lists information about the first file of each chapter *chap1.1*, *chap2.1*, etc.

Of these niceties, * is certainly the most useful to become familiar with. The others are frills, but worth knowing.

If the special meaning of *, **?**, etc. needs to be turned off, enclose the entire argument in single quotes as follows:

```
ls '?'
```

Some examples of this will be shown in the following paragraphs.

3.6 What's in a File Name, Continued

When the file called *junk* is first created, how does the system know that there is not another *junk* somewhere else, especially since the person in the next office could also be reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to that particular user. When you login, you are "in" your directory. Unless the user takes special action when creating a new file, the new file is made in the directory that the user is currently in. This is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in another (someone else's) directory.

The set of all files is organized into a (usually big) tree with your files located several branches into the tree. It is possible for you to "walk" around this tree and find any file in the system by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start at your present location and walk toward the root.

Try the latter first. The basic tool is the command **pwd**(1) (print working directory) which prints the name of the directory the user is currently in.

Although the details will vary according to the system the user is on, the **pwd**(1) command will print something like:

```
/usr/your_name
```

This message indicates that the user is currently in the directory *your_name*, which is in turn in the directory */usr*, which is in turn in the root directory called by convention just */*. (Even if it is not called */usr* on your system, the message will be something analogous. Recognize any differences between your machine's pathname and the standard setup and make the corresponding changes to the following command lines when appropriate.)

If user now types

```
ls /usr/your_name
```

the results should be exactly the same list of file names as obtained from a plain **ls**(1). With no arguments, **ls** lists the contents of the current directory. Given the name of a directory, it lists the contents of that directory.

Next, try using the following command:

```
ls /usr
```

This should print a long series of names, among which is your own login name *your_name*. On many systems, *usr* is a directory that contains the directories of all the normal users of the system.

The next step is to try the following:

```
ls /
```

The response should be something like this (although again the details may be different):

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

If *junk* is still in your directory, enter the following:

```
cat /usr/your_name/junk
```

The name

```
/usr/your_name/junk
```

is called the *pathname* of the file that is normally thought of as *junk*. The *pathname* represents the full name of the path as followed from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX operating system that anywhere an ordinary file name can be used the *pathname* can also be used.

This is not too exciting if all the files of interest are in your own directory; but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by entering the following:

```
pr /usr/your_name/chap*
```

Similarly, you can find out what files your neighbor has by entering:

```
ls /usr/neighbor
```

The "neighbor" just entered represents the login name of your neighbor. A copy of one of your neighbor's files can be made as follows:

```
cp /usr/neighbor/his_file your_file
```

If a file owner does not want someone else to have access to the owner's files or vice versa, privacy can be arranged. Each file and directory has read-write-execute (rwx) permissions for the owner, a group, and everyone else, which can be set to control access. See **ls**(1) and **chmod**(1) for details. Most users find openness of more benefit than privacy (most of the time).

As a final experiment with pathnames, try the following:

```
ls /bin /usr/bin
```

Do some of the names look familiar? When a program is run by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically does not find it), then in */bin* and finally in */usr/bin*. There is nothing magic about commands like **cat**(1) or **ls**(1), except that they have been collected into a couple of places to be easy to find and administer.

It is possible for two or more users to work regularly with common information in the same document. This common document should be divided up into several files. To prevent users from working in the same file at the same time, the users should be allowed to work only on specfied files. The files that make up this common document can be located in the directories of several users. These files can be combined into one document using the copy command [**cp**(1)] or the .so macro. If this common document is to be located in the same directory, the users can change the current working directory as follows:

```
cd full_path_name
```

Now you are ready to edit your specified files in this directory.

Another method of working on the same document is to locate the files in your friend's directory and login as your friend. Take into consideration that this defeats the accounting purpose of individual logins. If you are already logged in as yourself and want to work in a friend's files, change the current working directory as follows:

```
cd /usr/your_friend
```

Now when a file name is used in something like **cat**(1) or **pr**(1), the command refers to the file in your friend's directory. Changing directories does not affect any permissions associated with a file. If you can not access a file, get the owner to change permissions via **chmod**(1). Of course, if you forget what directory you are in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when writing your book, the user might want to keep all the text in a directory called *book*. A directory can be made using the **mkdir**(1) command. The *book* directory is made as follows:

```
mkdir book
```

The *book* directory can now be accessed to input chapters as follows:

```
cd book
```

If you logged in as yourself, the *pathname* of *book* is:

```
/usr/your_name/book
```

To remove the *book* directory, type:

```
rm book/*
rmdir book
  or
rm −r book
```

The **rm book/*** command removes all files in the *book* directory. The **rmdir book** command is then used to remove the empty directory. The *book* directory must be empty before the **rmdir** command will work. The **rm −r book** command recursively deletes the entire contents of the *book* directory and then removes the *book* directory itself.

The user can go up one level in the tree of files by entering:

```
cd ..
```

The ".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

3.7 Using Files for I/O Instead of the Terminal

Most of the commands used so far produce output on the terminal. Other commands, like the editor, take input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output.

As one example,

```
ls
```

makes a list of files on your terminal. But if the user enters

```
ls >filelist
```

a list of your files will be placed in the file *filelist* (which will be created if it does not already exist or overwritten if it does). The symbol > means "put the output on the following file rather than on the terminal". Nothing is produced on the terminal. As another example, the user could combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 >temp
```

Another symbol, that operates very much like > does, is >>. The >> means "add to the end of". That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate *f1*, *f2*, and *f3* to the end of whatever is already in *temp* instead of overwriting the existing contents. As with >, if *temp* does not exist, it will be created.

In a similar way, the symbol < means to take the input for a program from the following file instead of from the terminal. Thus, the user could make up a script of commonly used editing commands and put them into a file called *script*. The script could then be run on a file by entering:

```
ed file <script
```

Another example is using **ed** to prepare a letter in file *let*. The letter (file *let*) could then be sent to several people as follows:

```
mail adam eve mary joe <let
```

3.8 Pipes

One of the novel contributions of the UNIX operating system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes—a pipeline.

For example,

```
pr f g h
```

will print the files *f*, *g*, and *h*, beginning each on a new page. Instead of printing the files separately, the files can be printed together as follows:

```
cat f g h >temp
pr <temp
rm temp
```

This method is more work than necessary. To take the output of **cat** and connect it to the input of **pr**, use the following pipe:

```
cat f g h | pr
```

The vertical bar | means to take the output from **cat** which would normally have gone to the terminal and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program **wc**(1) counts the number of lines, words, and characters in its input; while the **who**(1) command prints a list of users currently logged on the system, one per access port.

Thus the command line

```
who | wc -l
```

tells how many people are logged on. And of course

```
ls | wc -l
```

counts your files.

Most programs that read from the terminal can read from a pipe instead. Most programs that write on the terminal can write on a pipe instead. There can be as many commands in a pipeline as desired.

Many UNIX operating system programs are written to take input from one or more files if file arguments are given. If no arguments are given, the programs will read from the terminal, and thus can be used in pipelines. One example using the **pr**(1) command to print files *a*, *b*, and *c* in three columns and in the order specified is as follows:

```
pr -3 a b c
```

But in

```
cat a b c | pr -3
```

the **pr** prints the information coming down the pipeline, still in three columns.

3.9 The Shell

The mysterious "**shell**" mentioned previously is actually the **sh**(1) command. The shell is the program that interprets what is typed as commands and arguments. The shell also looks after translating *, etc., into lists of file names, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, the user can run two programs with one command line by separating the commands with a semicolon. The shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

More than one program can run *simultaneously* if desired. This is beneficial when doing something time-comsuming, like using the editor script. The act of running programs in the background prevents waiting around for the results before starting something else. An example follows:

```
ed file <script &
```

The ampersand (&) at the end of a command line means "start this command running, then take further commands from the terminal immediately", that is, don't wait for it to complete. Thus the script will begin, but the user can do something else at the same time. Of course, to keep the output from interfering with what you are doing on the terminal, it would be better to enter

```
ed file <script >script.out &
```

which saves the output lines in a file called *script.out*.

When a command is initiated with **&**, the system replies with a number called the process number. Programs running simultaneously can be terminated as follows:

```
kill process_number
```

The process number is used to identify the command to be stopped. If you forget the process number, the **ps**(1) command will list the process number for all programs you are running. (Entering **kill 0** will kill all your processes.) If you are curious about other people, **ps −a** will provide information about all *active* programs that other users are running.

To start three commands that will execute in the order specified and in the background, enter the following:

```
(command_1; command_2; command_3) &
```

A background pipeline can be started as follows:

```
command_1 | command_2 &
```

Just as the editor or some similar program can get its input from a file instead of from the terminal, the shell can read a file to get commands. For example, suppose the user wants to perform a sequence of actions after every login such as:

- Set the tabs on the terminal
- Find out the date
- Find out who's on the system.

The three necessary commands to perform these actions [**tabs**(1), **date**(1), and **who**(1)] could be put in a file called *startup*. The *startup* file would then be run as follows:

```
sh startup
```

This instruction commands the machine to run the shell with the file *startup* as input. The effect is the same as typing the contents of *startup* on the terminal.

If this is to be a regular thing, the need to type **sh** every time can be eliminated by typing the following command only once:

```
chmod +x startup
```

To run the sequence of commands thereafter, the user only needs to enter:

```
startup
```

The **chmod**(1) command marks the file as being executable. The shell recognizes this and runs it as a sequence of commands.

If the user wants *startup* to run automatically for every login, create a file in your login directory called *.profile* and place in it the line "startup". Upon logging in, the shell gains control and executes the commands found in the *.profile* file. We will get back to the shell in the section on programming.

3.10 DOCUMENT PREPARATION

UNIX operating systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, etc. The **nroff** program is designed to produce output on terminals and line-printers. The **troff** (pronounced "tee-roff") program was designed to drive a phototypesetter, which produces very high quality output on photographic paper. This document was formatted with **troff**.

3.10.1 Formatting Packages

The basic idea of **nroff** and **troff**(1) is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there may be commands that specify how long lines are, whether to use single or double spacing, and the running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multicolumn output, etc. with little effort and without having to learn **nroff** and **troff.** These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

This section provides a brief description of the "memorandum macros" package known as **mm**(1). Formatting requests typically consist of a period and two uppercase letters, such as

```
.TL
```

which is used to introduce a title or

```
.P
```

to begin a new paragraph.

The text of a typical document is entered so it looks something like this:

```
.TL
title
.AU "author information"
.MT "memorandum type"
.P
Enter text ---
---
.P
More text ---
---
.SG "signature"
```

The lines that begin with a period are the formatting macro requests. For example, **.P** calls for starting a new paragraph. The precise meaning of **.P** depends on the output device being used (typesetter or terminal, for instance) and the publication the document will appear in. For example, −**mm** normally assumes that a paragraph is preceded by a space—one line in **nroff,** and one-half line in **troff,** and the first word is indented. These rules can be changed if desired, but they are changed by changing the interpretation of **.P**, not by retyping the document.

To actually produce a document in standard format using **−mm,** use the command

```
troff −mm files ...
```

for the typesetter and

```
nroff −mm files ...
```

for a terminal. The **−mm** argument tells **troff** and **nroff** to use the manuscript package of formatting requests. There are several similar packages; check with a local expert to determine which ones are in common use on your machine. The proper terminal filter for the terminal should be used in the command line. The terminal filter option is indicated by **−T** followed by the terminal type. The terminal types are known by various UNIX system utility calls found in section 1 of the *Sys5 UNIX User Reference Manual.*

3.10.2 Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the *Sys5 UNIX User Reference Manual* and check with UNIX operating system users for other possibilities.

Both **eqn**(1) and **neqn** (see **eqn** for more information) programs let you integrate mathematics into the text of a document in an easy-to-learn language that closely resembles the way you would speak it aloud.

For example, the **eqn** input

```
sum from i=0 to n x sub i ~=~ pi over 2
```

produces the output

The program **tbl(1)** provides an analogous service for preparing tables. The **tbl** program does all the computations necessary to align complicated columns with elements of varying widths.

The **spell**(1) program detects possible spelling mistakes in a document. The **spell** program compares the words in your document to a dictionary (stored in memory) and prints those words that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a good job.

The **grep**(1) program looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters *ing* in the files *chap**. The "**$**" indicated that the pattern to search for is at the end of the line, whereas a "^" indicates that the pattern to search for is at the beginning of a line. (It is almost always a good practice to put single quotes around the pattern to be searched for in case it contains characters like * or **$** that have a special meaning to the shell.) The **grep** program is often used to locate the misspelled words detected by the **spell** program.

The **diff**(1) program prints a list of the differences between two files, so that two versions of something can automatically be compared. This is a vast improvement over proofreading by hand.

The **wc**(1) program counts the words, lines, and characters in a set of files. The **tr**(1) program translates characters into other characters. For example, **tr** will convert uppercase to lowercase and vice versa. This translates uppercase into lowercase:

```
tr [A-Z] [a-z] <input >output
```

The **sort**(1) program sorts files in a variety of ways while **cxref**(1) makes cross-references. The **ptx**(1) program makes a permuted index (keyword-in-context listing). The **sed**(1) program provides many of the editing facilities of **ed** but can apply them to arbitrarily long inputs. The **awk**(1) program provides the ability to do both pattern matching and numeric computations and to conveniently process fields within lines. These programs are for more advanced users and are not limited to document preparation. Put them on your list of things to learn.

Most of these programs are either independently documented in the supplemental package like **eqn**(1) and **tbl**(1) in the DOCUMENTER'S WORKBENCH software option, or the programs are sufficiently simple enough so that the description in the *Sys5 UNIX User Reference Manual* is an adequate explanation.

3.10.3 Hints for Preparing Documents

Most documents go through several versions (always more than expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions simplify any editing needed later.

Keep the individual files of a document down to modest size, perhaps 10 to 15 thousand characters. Larger files edit more slowly. If a dumb mistake is made, it is better to clobber a small file than a big one. Split the files at natural boundaries in the document for the same reasons that you start each sentence on a new line.

The second aspect of making changes to documents easy is not to commit to the formatting details too early. One of the advantages of formatting packages is permitting format decisions to be delayed until the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or printed out on a line printer.

As a rule of thumb, a document should be produced in terms of a set of requests or commands (macros) for all but the most trivial jobs. The macros used should then be defined either by using one of the existing macro packages (the recommended way) or by defining your own **nroff** and/or **troff** macros. As long as the text is entered in some systematic way, it can always be cleaned up and formatted by a judicious combination of editing commands and macro definitions.

3.11 Programming

There will be no attempt made to teach any of the programming languages available, but a few words of advice are in order. One of the reasons why the UNIX operating system is a productive programming environment is that there is already a rich set of tools available. Facilities like pipes, input/output redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing a program completely from scratch.

The *Sys5 UNIX Programmer Reference Manual* contains the UNIX system programming utilities.

3.11.1 Shell Programming

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

cat ...	*collect the files*
|tr ...	*put each word on a new line*
|tr ...	*delete punctuation, etc.*
|sort	*into dictionary order*
|uniq	*discard duplicates*
|comm	*print words in text but not in dictionary*

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, the user could laboriously type:

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

The same job can be performed much more easily. One procedure is to type

```
ls chap* >temp
```

to get the list of file names into a file called *temp*. The *temp* file is then edited using global commands as follows:

```
1,$ s/^.*$/e &\
1p\
$p/
```

The results are written into the *script* file (1,$ w script) and then the following command is entered:

```
ed <script
```

This will produce the same output as the laborious hand typing. Another method is using shell loops to repeat a set of commands over and over again for a set of arguments as illustrated below:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable *i* to each file name in turn, then does the command. This command can be entered at the terminal or put in a file for later execution. Before the file can be executed, it may be necessary to change the mode by entering the following:

chmod +x *filename*

3.11.2 Programming with Shell

An option often overlooked by new users is that the **shell** is itself a programming language, with variables, control flow **if-else, while, for, case**, subroutines, and interrupt handling. Since there are many building-block programs, the user can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in section "AN INTRODUCTION TO SHELL" described later in this volume.

3.11.3 Programming in C

The C language is a reasonable choice of a programming language when undertaking anything substantial. Everything in the UNIX operating system is based on the C language. The system itself is written in C, as are most of the programs that run on the system. The C language is also an easy language to use once you get started. The C language is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how to do input/output and similar functions.

Most input and output in C is best handled with the standard input/output library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library. (Refer to Section 3 of the *Sys5 UNIX User Reference Manual.)*

The C programs that do not depend too much on the special features of the UNIX operating system (such as pipes) can be moved to other computers that have C compilers.

There are a number of supporting programs that go with C. The **lint**(1) program checks C programs for potential portability problems and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file), the **make**(1) program allows you to specify the dependencies among the source files and the processing steps needed to make a new version. The program then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger **sdb**(1) program is useful for digging through the dead bodies of C programs but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited statistical service, so a user can find where programs spend their time executing and what parts of a program are worth optimizing. Compile the programs with the **−p** option; after the test run, use the **prof**(1) command to print a program execution profile. The command **time**(1) will give the gross run-time statistics of a program, but the times are not very accurate or reproducible.

3.11.4 Other Languages

If Fortran *must* be used, there are two possibilities— Fortran 77 and ratfor. The user might consider **ratfor** which provides decent control structures and free-form input that characterize C, yet permits the writing of code that is also portable to other environments. Bear in mind that UNIX operating system Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **prof**(1), etc., are all virtually useless with Fortran programs. If there is a Fortran 77 compiler on your system, it may be a viable alternative to **ratfor** and has the nontrivial advantage that it is compatible with the C language and related programs. (The ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires translating a language into a set of actions or another language, the user is in effect building a compiler, though probably a small one. In that case, the **yacc**(1) compiler-compiler is recommended for use, which aids in developing a compiler quickly.

The **lex**(1) lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself or as a front end to recognize inputs for a **yacc-based** program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later.

4. TUTORIAL-TEXT-EDITOR

Almost all text input on the UNIX operating system is done with the standard text editor **ed**(1). This is a tutorial guide to help beginners get started with text editing.

Although this guide does not cover everything about the UNIX operating system, it does discuss enough for most user's day-to-day needs. This includes printing, appending, changing, deleting, moving, and inserting entire lines of text; reading and writing files; context searching and line addressing; substituting; global changing; and using some special characters for easier editing.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX V Administrator Reference Manual*.· Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX V User Reference Manual*. Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX V Programmer Reference Manual*.

4.1 GENERAL

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

This tutorial is meant to simplify learning **ed**. The recommended way to learn **ed** is to read this document, while simultaneously using **ed** to follow the examples, then to read the description in Section 1 of the *Sys5 UNIX V User Reference Manual*. Getting advice from experienced UNIX operating system users and experimenting with **ed** are also useful.

Do the exercises! The exercises illustrate techniques not completely discussed in the actual text. A summary at the end of this section summarizes the commands.

4.1.1 Disclaimer

This is a tutorial introduction and guide only. For this reason, no attempt is made to cover more than a part of the facilities that **ed** offers (although this fraction includes the most useful and frequently used facilities). Also, there is not enough space to explain the basic UNIX operating system procedures. It is assumed that the user knows how to log on to the UNIX operating system and has a vague understanding of what a UNIX operating system file is. For more on the UNIX operating system facilities, refer to the section, "Basics For Beginners".

The user must also know what character to type as the end-of-line character on the user's particular terminal. This character is the RETURN or newline character (key) on most terminals. Hereafter reference to the end-of-line character, whatever it is, will be referred to as RETURN.

4.2 GETTING STARTED

Assume that the user has logged in to a UNIX operating system and it has just printed the *prompt character*, usually a

```
$
```

The easiest way to invoke **ed** is to type:

```
ed  (followed by a RETURN)
```

You (the user) are now ready to go. The **ed** program is waiting to be told what to do.

4.2.1 Creating Text—The Append Command "a"

As your first problem, suppose some text is to be created starting from scratch. Perhaps the very first draft of a document or paper is to be entered. Normally, it will have to start somewhere and undergo modifications (editing) later. This part will describe how to enter some text to get a file of text started. How to make changes and corrections to the text is described later.

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

First a bit of terminology. In **ed** jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if desired, or simply as the information that is to be edited. In effect the buffer is like the piece of paper on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells **ed** what to do to the text by typing instructions called "commands." Most commands consist of a single lowercase letter. Each command is typed on a separate line. (Sometimes the command is preceded by information about the line or lines of text to be affected—these will be described below.) The **ed** text editor makes no response to most commands—there is no prompting or response messages like "ready". (This silence is preferred by experienced users.

The first command is append, written as the letter

```
a
```

on a command line all by itself. It means "append (or add) text lines to the buffer as I type them in."

Appending is rather like writing fresh material on a piece of paper. So to enter lines of text into the buffer, just type an

```
a
```

followed by a RETURN and the lines of text desired, like this:

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

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

After the append command has been used, the buffer will contain the following three lines:

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

The **a** and the "." are not there because they are not text.

To add more text to what already exists, just issue another **a** command and continue typing.

4.2.2 Error Messages (?)

If at any time the user makes an error in the commands typed into **ed**, the text editor will tell the user by typing the following:

```
?
```

This is about as cryptic as it can be, but with practice, the user can usually figure out the goof. The user can get a brief explanation of the error by typing

```
h
```

The **help** command gives a short error message that explains the reason for the most recent **?** diagnostic.

4.2.3 Writing Text File—The Write Command "w"

It is likely that you will want to save your text for later use. To write out the contents of the buffer onto a file, use the write command

```
w
```

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

```
w junk
```

Leave a space between **w** and the file name. The **ed** program will respond by printing the number of characters it wrote out. In this case, **ed** would respond with:

```
68
```

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

4.2.4 Leaving ed—The Quit Command "q"

To terminate a session with **ed**, first save your text by writing it onto a file using the **w** (write) command, and then type the **q** (quit) command:

```
q
```

The system will respond with the prompt character:

```
$
```

At this point your buffer vanishes, with all its text, which is why the user would want to write before quitting. Actually **ed** will print the character

```
?
```

if the user tries to quit without writing. At this point, the user writes if desired; if not, another **q** will get you out regardless and will not save the text in the buffer. EXERCISES—TRY THEM!

4.3 EXERCISES

4.3.1 EXERCISE 1

Enter **ed** and create some text using the append command **a**

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

Note that no system prompt appears while in the text editor. Do not forget to write the text into memory with the write command **w**. Write it into memory using the **w** command. Then leave **ed** with the **q** command and print the file to see that everything worked. To print a file, enter

```
pr filename
   or
cat filename
```

in response to the prompt character ($). Try both.

4.3.1.1 Reading Text File— Edith Command "e"

A common way to get text into the buffer is to read it from another file in the file system. This is what you do to edit text that you saved with the **w** command in a previous session. The edit command

```
e
```

retrives the entire contents of a file into the buffer.

So if the user had saved the three lines "Now is the time", etc., with a **w** command in an earlier session, the edit command

```
e junk
```

would place the entire contents of the file *junk* into the buffer and respond with a number

```
68
```

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

If the **e** command is used to read a file into the buffer, then the user does not need to use a file name after a subsequent **w** command; **ed** remembers the last file name used in an **e** command, and **w** will write on this file. Thus a good practice to follow is:

```
ed
e filename
[editing session]
.
w
q
```

This way, the user can simply enter **w** from time to time and be secure in the knowledge that if the user got the file name right at the beginning, the user is writing into the proper file each time. Note that after each edit command **e** or each write command **w** the number of characters is returned by **ed**. The user can find out at any time what file name **ed** is remembering by typing the file command **f**. In this example, if you typed

```
f
```

ed would reply

```
junk
```

4.3.1.2 Reading Text—The Read Command "r"

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

```
r junk
```

will read the file *junk* into the buffer. The command appends the file specified to the end of whatever file is already in the buffer. So if you do a read after an edit command such as

```
e junk
r junk
```

the buffer will contain *two* copies of the orginal text as follows:

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

Like the **w** and **e** commands, **r** prints the number of characters read in after the reading operation is complete. Generally speaking, **r** is much less used than **e**.

The read command **r** may also be used to read a file external to the buffer into the file in the buffer. While in **ed** and at the current line, enter the command

```
.r filename
```

and filename will be read into the file (already in the buffer) immediately after the current line. None of the file in the buffer is destroyed, rather the external file *filename* has been read into and been combined with the file already in the buffer. The file that was read remains in filename also. You only copied it. Notice the difference between "r" and ".r".

4.3.2 EXERCISE 2

Experiment with the **e** command—try reading and printing various files. The user may get an error **?name** where *name* is the name of a file. This means that the file does not exist. Some typical causes of getting an empty file are spelling the file name wrong or perhaps trying to read or write a particular file which your permissions will not allow. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

4.3.2.1 Printing Buffer Contents—The Print Command "p"

To print or list the contents of the buffer (or parts of it) on the terminal, use the print command **p**. This is done as follows. Specify the line numbers where printing is to begin and end. These numbers have a comma between the beginning number and the ending number, i.e., "beginning line number, ending line number p". Thus to print the first ten lines of the contents of any buffer (i.e., lines 1 through 10), type:

```
1,10p    (prints lines 1 through 10)
```

The **ed** will respond by printing the specified starting line (1) through the specified ending line (10).

Suppose it is desirable to print *all* the lines in the buffer. You could use "1,30p" as above if it is known there were exactly 30 lines in the buffer. But in general, it is not known how many lines there are, so what can be used for the ending line number? The **ed** program provides a shorthand symbol for "line number of the *last line* in the buffer"—the dollar sign **$**. To print all the lines in the buffer, use it this way:

```
1,$p    (Prints all lines in buffer)
              or
,p      (Prints all lines in buffer also)
```

This will print *all* the lines in the buffer (line 1 through the last line). The "1,$p" can be abbreviated "$,p". To stop the printing before the last line is printed, push the **DEL** key or the **DELETE** (or equivalent) key on the terminal. The **ed** program will respond

```
?
```

and wait for the next input command.

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

```
$,$p
```

but **ed** lets you abbreviate this to

```
$p
```

Any *single* line can be printed by typing the line number followed by a **p**. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, **ed** lets you abbreviate even further. You can print any single line by typing *just* the line number—no need to type the letter **p**. So by entering

```
$
```

ed will print the last line of the buffer. Entering a single line number will print that line only.

It is also possible to use **$** in combinations like

```
$-5,$p
```

which prints the *last five* lines of the buffer. This helps to determine the end of the contents of the buffer when more is to be entered.

4.3.3 EXERCISE 3

Create some text using the **a** command and experiment with the **p** command. The user will find, for example, that line 0 or a line beyond the end (last line) of the buffer cannot be printed. Attempts to print a buffer in reverse order by entering

```
3,1p
```

will *not* work.

4.3.3.1 The Current Line "." or Dot

Suppose the buffer still contains the six lines of text (as in Exercise 1), and the following was entered

```
1,3p
```

and **ed** has printed the three lines.

Try typing just

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

This will print

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

which is the third line of the buffer. In fact it is the last (most recent) line that anything was done to. (The line just printed!) The **p** command can be repeated without line numbers, and it will continue to print line 3.

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

```
.       (Pronounced "dot")
```

Dot is a line number in the same way that **$** is. Dot means exactly "the current line", or loosely, "the line something was done to most recently." The dot can be used in several ways—one possibility is to enter:

```
.,$p
```

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

Some commands change the value of dot, while others do not. The print command **p** sets dot to the number of the last line printed; the last command entered (.,$p) will set both "." and **$** to the last line in the buffer (line 6).

Dot is most useful when used in combinations like:

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

This means "print the next line" and is a handy way to step slowly through a buffer. The user can also enter

```
.−1     (or .−1p)
```

which means "print the line *before* the current line." This enables stepping through the buffer backwards if desired. Another useful one is something like

```
.−3,.−1p
```

which prints the previous three lines.

Do not forget that all of these change the value of dot. The user can find out what dot is at any time by typing

```
.=      (dot line number is ?)
```

The **ed** program will respond by printing the value (line number) of dot.

Let us summarize some things about the **p** command and dot. Essentially, **p** can be preceded by 0, 1, or 2 line numbers (for our example). If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given with or without the letter **p**, it prints that line and sets dot there. If there are two line numbers separated by a comma, it prints all the lines in that range from the first number to the last number, and sets dot to the last line printed. If two line numbers are specified, the first cannot be bigger than the second (refer to the beginning of "EXERCISE 3").

Typing a single RETURN will cause printing of the next line—it is equivalent to:

```
.+1p
```

Try it. Typing a ^ is equivalent to typing the minus −. It can be used in multiples, as ^^^, which will move the current line or dot line backwards three lines from the current line. The "−" or the "^" can be considered equivalent to "−1p" since either moves the dot back one line.

4.3.3.2 Deleting Lines—The Delete Command "d"

Suppose three extra lines in the buffer are not needed. They may be removed by use of the delete command:

```
d
```

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

```
starting line, ending line d
```

Thus the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left that can be checked by using:

```
1,$p
```

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

The delete command **d** and the print command **p** may be used together thus

```
dp
```

which deletes the current line, prints the following line, and sets dot to the line printed.

4.3.4 EXERCISE 4

Experiment with **a**, **e**, **r**, **w**, **p**, and **d** until you become familiar with their use. While experimenting, also use "dot", **$**, and line numbers to understand their use.

When you start to feel adventurous, try using line numbers with **a**, **r**, and **w** as well. The user will find that **a** will append lines *after* the line number that you specify (rather than after dot); **r** reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and **w** will write out exactly the lines specified, not necessarily the whole buffer. These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering:

```
0r filename
```

Lines can be entered at the beginning of the buffer by using:

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

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

```
.
w
```

4.3.4.1 Modifying Text—The Substitute Command "s"

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

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. The substitute command is used for correcting spelling mistakes and typing errors.

Suppose that, because of a typing error, line 1 says

```
Now is th time
```

notice the *e* has been left off. The **s** command can be used to fix this as follows:

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

This says: in line 1, substitute for the characters *th* the characters *the*. Since **ed** will not print the result automatically, enter

```
p
```

to verify that the substitution worked, and you should get

```
Now is the time
```

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

The general way to use the substitute command is

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

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

Thus the following can be entered

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

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

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

```
s/something/something else/p
```

which makes some correction on the current line and then prints it (current line) to make sure it worked out right. If it did not, you can try again. Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any substitute command.

It is also legal to say

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

which means change the first string of characters (....) to *nothing*, i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words.

For instance, if the buffer contained

```
Nowxx is the time
```

this can be corrected by entering

```
s/xx//p
```

to get

```
Now is the time
```

Notice that // (two adjacent slashes) means "no characters" *not* a blank. There *is* a difference! (See "Context Searching" under "EXERCISE 5" for another meaning of "//").

4.3.5 EXERCISE 5

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

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

which results in the following:

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string.

All occurrences can be changed by adding a **g** (for "global") command to the **s** command, like this:

```
s/.../.../gp
```

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

If strange results are produced by inputting

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

read the part under "Special Characters" in this section.

4.3.5.1 Context Searching "/...../"

When the substitute command is mastered, you may move on to another highly important feature of **ed**(1)—context searching.

Suppose the original three lines of text in the buffer is as follows:

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

Suppose the word *their* is to be changed to *the*. How is the line that contains *their* located? With only three lines in the buffer, it is pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines and you had been making changes, deleting and rearranging lines, etc., you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context (unique text) on it.

The way to say "search for a line that contains this particular string of characters" or "unique text" is to type:

```
/string of characters to find/
```

For example, the **ed** expression

```
/their/
```

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

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

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

```
?
```

Otherwise, it prints the line it found.

The search for the desired line *and* the substitution can be done together, like this

```
/their/s/their/the/p
```

which will yield

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

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

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

Suppose the buffer contains the three familiar lines

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

Then the **ed** line numbers

```
/Now/+1
/good/
/party/-1
```

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

```
/Now/+1s/good/bad/
     or
/good/s/good/bad/
     or
/party/-1s/good/bad/
```

The choice is dictated only by convenience. All three lines could be printed by entering

```
/Now/,/party/p
    or
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you do not know how many lines are involved. The basic rule is: a context search expression is the *same* as a line number, so it can be used wherever a line number is needed.

4.3.6 EXERCISE 6

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

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

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

If funny results are obtained with any of the characters

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

read the part in this section on "Special Characters".

The **ed** program provides a short method for repeating a context search for the same string. For example, the **ed** line number

```
/string/
```

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

```
//
```

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

```
/string1/s//string2/
```

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

```
??
```

means "scan backwards for the same expression."

4.3.6.1 Change and Insert Commands "c" and "i"

This section discusses the change command

```
c
```

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

```
i
```

which is used for inserting a group of one or more lines immediately before the current line.

"Change", written as

```
c
```

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change the first line (.+1) past the current line through the last line ($) of a file to something else, type

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

The lines typed between the **c** command and the '.' (dot) command will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors.

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

"Insert" is similar to append—for instance

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

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

4.3.7 EXERCISE 7

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

```
starting-line,ending-line d
i
...text...
.
```

is almost the same as

```
starting-line,ending-line c
...text...
.
```

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

Experiment with the append command **a** and the insert command **i** to see that they are similar but not the same. You will observe that

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

appends *after* the given line, while

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

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, **a** appends after line dot, and **c** changes line dot.

4.3.7.1 Moving Text Around—The Move Command "m"

The move command **m** is used for cutting and pasting—it allows a group of lines to be moved from one place to another in the buffer. Suppose the first three lines of the buffer are to be placed at the end of the buffer instead of at the beginning. This could be performed by entering:

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

(Do you see why?) This method will work, but it is a lot easier using the **m** command as follows:

```
1,3m$
```

The general case is:

```
starting-line,ending-line m after this line
```

Notice that there is a third line to be specified—the line after which the other lines are to be moved. 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.
```

the two paragraphs could be reversed 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.

4.4 THE GLOBAL COMMANDS

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

```
g/peling/p
```

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

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

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

Compare this to

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

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

There may be several commands used in conjunction with the **g** command, but every line except the last must end with a backslash "\". For example:

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

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

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

```
v/ /d
```

deletes every line that does not contain a blank.

4.5 SPECIAL CHARACTERS

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

```
/x.y/
```

means "a line with an **x**, *any character*, and a **y**", *not* just "a line with an **x**, a period, and a **y**."

The following is a complete list of the special characters that can cause trouble:

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

Warning: **The backslash character "\" is special to "ed". For safety's sake, avoid it where possible.**

If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

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

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

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

```
/^string/
```

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

```
string
```

but not

```
the string...
```

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

The input

```
/string$/
```

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

```
/^string$/
```

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

```
/^.$/
```

finds a line containing exactly one character.

The character "**.**", as we mentioned above, matches anything. For example, the input

```
/x.y/
```

matches any of the following:

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

This is useful in conjunction with "*" which is a repetition character. The "a*" is a shorthand input for "any number of a's" therefore ".*" matches any number of anythings.

For example, input

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

which changes an entire line, or

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

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

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

```
/[0123456789]/
```

matches any single digit—any one of the characters inside the braces will cause a match. This can be abbreviated to

```
[0-9]
```

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

Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. One tedious method is just to retype the line. Another method is to enter

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

using your knowledge of "^" and "$". But the easiest way uses the "&" as follows:

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

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

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

to produce

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

You do not have to match the whole line, of course. If the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

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

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

Inputting

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

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

4.6 SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of the **ed** text editor commands is the command **name**, perhaps preceded by one or two line numbers. In the case of the edit command **e**, the read command **r**, and the write command **w**, the command **name** is also followed by a *file name*. Normally, only one command is allowed to be entered per line, but a print command **p** may follow any other command (except for the edit command **e**, the read command **r**, the write command **w**, and the quit command **q**).

a	*Append*, adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a dot "." is typed at the beginning (first character) of a new line. Dot is set to the last line appended.
c	*Change* the specified lines to the new text which follows. Entering new lines is terminated by a dot "." as with **a**. If no lines are specified, the current line (dot) is replaced. Dot is set to last line changed.

d	*Delete* the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line unless **$** is specified, in which case dot is set to the last line, **$**.
e	*Edit* new file. Any previous contents of the buffer are thrown away, so issue a write command **w** beforehand.
f	Print the remembered *file* name. If a name follows **f**, the remembered name will be set to it.
g	The *global* command **g/---/commands** will execute the commands on those lines that contain "---".
i	*Insert* lines before the specified line or the current line (dot line) until a "." is typed at the beginning of a new line. Dot is set to last line inserted.
m	*Move* lines specified to the line named after **m**. Dot is set to the last line moved.
n	Print the *number* of the addressed line(s) followed by a tab and the line itself.
p	*Print* specified lines. If none specified, print line dot. A single line number is equivalent to "line number". A single RETURN prints the next line, i.e., the dot plus one line, ".+1".
q	The *quit* command exits from **ed**. It wipes out all text in buffer if you give it twice in a row without first giving a write command **w**.
r	*Read* a file into buffer (at end unless specified elsewhere). Dot set to last line read. If **.r filename** is used, the filename is read into the buffer immediately after the dot line.
s	The **s/string1/string2/** command is used to *substitute* the characters "string1" into "string2" in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place; if no substitution took place, dot is not changed. The command **s** changes only the first occurrence of "string1" on a line; to change all occurrences on a line, type a **g** after the final slash.

v	The *exclude* command **v/---/commands** executes commands only on those lines that do *not* contain "---".
w	The *write* command writes out the buffer contents onto a file. Dot is not changed.
.=	The ".=" causes the printout of the current line number. The *dot value* prints the line number of the current line (dot line). The "=" by itself prints the value of the last line in the file.
!	The "!" is a *temporary escape* command. The line "command-line" causes "command-line" to be executed as a UNIX operating system command.
/-----/	The *context search* command searches for next line which contains this string of characters "----" and prints it. Dot is set to the line where string was found. Search starts at line ".=1" then wraps around from the last line "$" to line "1" and continues to dot (the current line) if necessary.
?-----?	Performs *context search* in reverse direction. Starts search at the previous line ".−1", scans to line 1, wraps around to the last line "$", and scans back to the current line (dot line) if necessary.

5. vi

This document provides a quick introduction to *vi* (pronounced *vee-eye*). You should be running *vi* on a file you are familiar with while you are reading this. An appendix lists each character and any special meanings this character has in *vi.*

5.1 Specifying Terminal Type

Before you can start *vi,* you must tell the system what kind of terminal you are using. Here is a (necessarily incomplete) list of terminal type codes. If your terminal does not appear here, you should consult with one of the staff members on your system to find out the code for your terminal. If your terminal does not have a code, one can be assigned and a description for the terminal can be created.

Code	Full name	Type
2621	Hewlett-Packard 2621A/P	Intelligent
2645	Hewlett-Packard 264x	Intelligent
act4	Microterm ACT-IV	Dumb
act5	Microterm ACT-V	Dumb
adm3a	Lear Siegler ADM-3a	Dumb
adm31	Lear Siegler ADM-31	Intelligent
c100	Human Design Concept 100	Intelligent
dm1520	Datamedia 1520	Dumb
dm2500	Datamedia 2500	Intelligent
dm3025	Datamedia 3025	Intelligent
fox	Perkin-Elmer Fox	Dumb
h1500	Hazeltine 1500	Intelligent
h19	Heathkit h19	Intelligent
i100	Infoton 100	Intelligent
mime	Imitating a smart act4	Intelligent
t1061	Teleray 1061	Intelligent
vt52	Dec VT-52	Dumb

Suppose, for example, that you have a Hewlett-Packard HP2621A terminal. The code used by the system for this terminal is '2621'. In this case you can use one of the following commands to tell the system the type of your terminal:

```
% setenv TERM 2621
```

This command works with the shell *csh* on all Plexus systems. If you are using the standard Sys5 shell, give the commands

```
        $ TERM=2621
        $ export TERM
```

If you want to arrange to have your terminal type set up automatically when you log in, you can use the *tset* program. For example, if you dial in on a *mime*, but often use hardwired ports, a typical line for your *.login* file (if you use *csh*) would be

```
        setenv TERM `tset --d mime`
```

or for your *.profile* file (if you use *sh*)

```
        TERM=`tset --d mime`
```

Tset knows which terminals are hardwired to each port and needs only to be told that when you dial in you are probably on a *mime*. *Tset* is usually used to change the erase and kill characters, too.

5.2 Editing a File

After telling the system what kind of terminal you have, you should make a copy of a file you are familiar with, and run *vi* on this file. Pick a file that is longer than one screenful (usually 24 lines) so you can see the effect of scrolling and other commands. Give the command

```
        % vi name
```

replacing *name* with the name of the copy file you just created. The screen should clear and the text of your file should appear on the screen.

5.3 The Editor's Copy: the Buffer

The editor does not directly modify the file you are editing. Rather, the editor makes a copy of this file, in a place called the *buffer,* and remembers the file's name. You do not affect the contents of the file unless and until you write the changes you make back into the original file.

0. If something else happens, you may have given the system an incorrect terminal type code. In this case, the editor may have just made a mess out of your screen (your file is probably okay). This sort of mess happens when the editor sends control codes for one kind of terminal to some other kind of terminal. To re-start, hit the keys **:q** (colon and the q key) and then hit the RETURN key. This should get you back to the command level interpreter. Figure out what you did wrong (ask someone else if necessary) and try again. You may also have typed the wrong file name and the editor just printed an error diagnostic. In this case you should follow the above procedure for getting out of the editor, and try again. If the editor doesn't seem to respond to the commands that you type here, try sending an interrupt to it by hitting the DEL or RUB key on your terminal, and then hitting the **:q** command again, followed by a carriage return.

5.4 Notational Conventions

In our examples, input that must be typed exactly as written here is presented in **bold face**. Text that should be replaced with appropriate input is given in *italics*. We represent special characters in SMALL CAPITALS.

5.5 Arrow Keys

The editor command set is independent of the terminal you are using. On most terminals with cursor positioning keys, these keys also work within the editor. If you don't have cursor positioning keys, or even if you do, you can use the **h j k** and **l** keys as cursor positioning keys (these are labeled with arrows on an *adm3a).**

(Particular note for the HP2621: on this terminal the function keys must be *shifted* to send to the machine; otherwise they only act locally. Unshifted use leaves the cursor positioned incorrectly.)

5.6 Special Characters: ESC, CR and DEL

Several of these special characters are very important, so be sure to find them right now. Look on your keyboard for a key labeled ESC or ALT. It should be near the upper left corner of your terminal. Try hitting this key a few times. The editor rings the bell to indicate that it is in a quiescent state. Partially formed commands are canceled by ESC, and when you insert text in the file you end the text insertion with ESC. This key is a fairly harmless one to hit, so you can just hit it if you don't know what is going on until the editor rings the bell.

The CR or RETURN key is important because it is used to terminate certain commands. It is usually at the right side of the keyboard, and is the same command used at the end of each shell command.

Another very useful key is the DEL or RUB key, which generates an interrupt, telling the editor to stop what it is doing. It is a forceful way of making the editor listen to you, or to return it to the quiescent state if you don't know or don't like what is going on. Try hitting the '/' key on your terminal. This key is used when you want to specify a string to be searched for. The cursor should now be positioned at the bottom line of the terminal after a '/' printed as a prompt. You can get the cursor back to the current position by hitting

0. * As we will see later, *h* moves back to the left (like control-h, which is a backspace), *j* moves down (in the same column), *k* moves up (in the same column), and *l* moves to the right.

0. On smart terminals where possible, the editor quietly flashes the screen rather than ringing the bell.

the DEL or RUB key; try this now.* From now on we will simply refer to hitting the DEL or RUB key as "sending an interrupt."**

The editor often echoes your commands on the last line of the terminal. If the cursor is on the first position of this last line, then the editor is performing a computation, such as computing a new position in the file after a search or running a command to reformat part of the buffer. When this is happening you can stop the editor by sending an interrupt.

5.7 Getting out of the Editor

After you have worked with this introduction for a while, and you wish to do something else, you can give the command **ZZ** to the editor. This writes the contents of the editor's buffer back into the file you are editing, if you made any changes, and then leaves the editor. You can also end an editor session by giving the command **:q!**CR; this is a dangerous but occasionally essential command that ends the editor session and discards all your changes. You need to know about this command in case you change the editor's copy of a file you wish only to look at. Be very careful not to give this command when you really want to save the changes you have made.

5.8 Moving around in the File

5.8.1 Scrolling and Paging

The editor has a number of commands for moving around in the file. The most useful of these is generated by hitting the control and D keys at the same time, a control-D or '^D'. We will use this two character notation for referring to these control keys from now on. You may have a key labeled '^' on your terminal. This key will be represented as '↑' in this document; '^' is exclusively used as part of the '^x' notation for control characters.

As you know now if you tried hitting **^D**, this command scrolls down in the file. The **D** thus stands for down. Many editor commands are mnemonic and this makes them much easier to remember. For instance the command to scroll up is **^U**. Many dumb terminals can't scroll up at all, in which case hitting **^U** clears the screen and refreshes it with a line which is farther back in the file at the top.

---

0. * Backspacing over the '/' also cancels the search.
0. ** On some systems, this interruptibility comes at a price: you cannot type ahead when the editor is computing with the cursor on the bottom line.
0. All commands that read from the last display line can also be terminated with a ESC as well as an CR.
0. If you don't have a '^' key on your terminal then there is probably a key labeled '↑'; in any case these characters are one and the same.

If you want to see more of the file below where you are, you can hit **^E** to expose one more line at the bottom of the screen, leaving the cursor where it is. The command **^Y** (which is hopelessly non-mnemonic, but next to **^U** on the keyboard) exposes one more line at the top of the screen.

Other ways to move around in the file include the keys **^F** and **^B**, which move forward and backward a page, keeping a couple of lines of continuity between screens.

Notice the difference between scrolling and paging. If you are trying to read the text in a file, hitting **^F** to move forward a page leaves you only a little context to look back at. Scrolling on the other hand leaves more context, and happens more smoothly. You can continue to read the text as scrolling is taking place.

5.8.2 Searching, Goto, and Previous Context

Another way to position yourself in the file is by giving the editor a string to search for. Type the character / followed by a string of characters terminated by CR. The editor positions the cursor at the next occurrence of this string. Try hitting **n** to then go to the next occurrence of this string. The character **?** searches backwards from where you are, and is otherwise like /.

If the search string you give the editor is not present in the file, the editor prints a message to that effect on the last line of the screen, and the cursor is returned to its initial position.

If you wish the search to match only at the beginning of a line, begin the search string with an ↑. To match only at the end of a line, end the search string with a **$**. Thus /↑**search**CR searches for the word 'search' at the beginning of a line, and /**last$**CR searches for the word 'last' at the end of a line.*

The command **G**, when preceded by a number, positions the cursor at that line in the file. Thus **1G** moves the cursor to the first line of the file. If you give **G** no count, it moves to the end of the file.

0. These searches normally wrap around the end of the file, and thus find the string even if it is not on a line in the direction you search provided it is anywhere else in the file. You can disable this wraparound in scans by giving the command **:se nowrapscan**CR, or more briefly **:se nows**CR.

0. *Actually, the string you give to search for here can be a *regular expression* in the sense of the editors *ex*(1) and *ed*(1). If you don't wish to learn about this yet, you can disable this more general facility by putting the command **:se nomagic**CR; in EXINIT in your environment. (More about *EXINIT* later.)

If you are near the end of the file, and the last line is not at the bottom of the screen, the editor places only the character ' ' on each remaining line. This indicates that the last line in the file is on the screen; that is, the ' ' lines are past the end of the file.

You can find out the state of the file you are editing by typing a ^**G**. The editor shows you the name of the file you are editing, the number of the current line, the number of lines in the buffer, and what percentage of the buffer you have traversed. Try doing this now, and remember the number of the line you are on. Give a **G** command to get to the end. You can get back to your previous position by using the command `` (two back quotes). Try giving a search with / or **?** and then a `` to get back to where you were. If you accidentally hit **n** or any command that moves you far away from a context of interest, you can quickly get back by hitting ``.

5.8.3 Moving around on the Screen

Now try just moving the cursor around on the screen. If your terminal has arrow keys (4 or 5 keys with arrows going in each direction) try them and convince yourself that they work. (On certain terminals, they won't.) If you don't have working arrow keys, you can always use **h**, **j**, **k**, and **l**. Experienced users of *vi* prefer these keys to arrow keys, because they are usually right underneath their fingers.

Hit the + key. Each time you do, notice that the cursor advances to the next line in the file, at the first non-white position on the line. The – key is like + but goes the other way.

These are very common keys for moving up and down lines in the file. Notice that if you go off the bottom or top with these keys then the screen scrolls down (and up if possible) to bring a line at a time into view. The RETURN key has the same effect as the + key.

Vi also has commands to take you to the top, middle and bottom of the screen. **H** takes you to the top (home or highest) line on the screen. Try preceding it with a number as in **3H**. This takes you to the third line on the screen. Many *vi* commands take preceding numbers and do interesting things with them. Try **M**, which takes you to the middle line on the screen, and **L**, which takes you to the last (or lowest) line on the screen. **L** also takes counts, thus **5L** takes you to the fifth line from the bottom.

5.8.4 Moving within a Line

Now try picking a word on some line on the screen, not the first word on the line. Move the cursor using RETURN and – to be on the line where the word is. Try hitting the **w** key. This advances the cursor to the next word on the line. Try hitting the **b** key to back up words in the line. Try the **e** key, which advances you to the end of the current word rather than to the beginning of the next word. Also try SPACE (the space bar), which moves right one character and the BS (backspace or ^**H**) key, which moves left one character. The key **h** works as ^**H** does and is useful if you don't have a BS key. (Also, as noted just above, **l** moves to the right.)

If the line has punctuation in it, you may notice that that the **w** and **b** keys stop at each group of punctuation. You can also go back and forward by words without stopping at punctuation by using **W** and **B** rather than the lower case equivalents. Think of these as bigger words. Try these on a few lines with punctuation to see how they differ from the lower case **w** and **b**.

The word keys wrap around the end of line, rather than stopping at the end. Try moving to a word on a line below where you are by repeatedly hitting **w**.

5.8.5 Summary

SPACE	advance the cursor one position
^**B**	backwards to previous page
^**D**	scrolls down in the file
^**E**	exposes another line at the bottom (v3)
^**F**	forward to next page
^**G**	tell what is going on
^**H**	backspace the cursor
^**N**	next line, same column
^**P**	previous line, same column
^**U**	scrolls up in the file
^**Y**	exposes another line at the top (v3)
+	next line, at the beginning
–	previous line, at the beginning
/	scan for a following string forwards
?	scan backwards
B	back a word, ignoring punctuation
G	go to specified line, last default
H	home screen line
M	middle screen line
L	last screen line
W	forward a word, ignoring punctuation
b	back a word
e	end of current word
n	scan for next instance of / or **?** pattern
w	word after this word

5.8.6 View

If you want to use the editor to look at a file, rather than to make changes, invoke it as *view* instead of *vi*. This sets the *readonly* option which prevents you from accidently overwriting the file.

5.9 Making Simple Changes

5.9.1 Inserting

One of the most useful commands is the **i** (insert) command. After you type **i**, everything you type until you hit ESC is inserted into the file. Try this now; position yourself to some word in the file and try inserting text before this word. If you are on an dumb terminal it may seem, for a minute, that some of the characters in your line have been overwritten, but they reappear when you hit ESC.

Now try finding a word which can, but does not, end in an 's'. Position yourself at this word and type **e** (move to end of word), then **a** for append and then '**s**ESC' to terminate the insertion. This sequence of commands (**eas**ESC) can be used to pluralize a word easily.

Try inserting and appending a few times to make sure you understand how this works; **i** placing text to the left of the cursor, **a** to the right.

You may often want to add new lines before or after some specific line in the file. Find a line where this makes sense and then give the command **o** to create a new line after the line you are on, or the command **O** to create a new line before the line you are on. After you create a new line in this way, all text you type up to an ESC is inserted. This may amount to many lines or just one (see below).

Many related editor commands are invoked by the same letter key and differ only in that one is given by a lower case key and the other is given by an upper case key. In these cases, the upper case key often differs from the lower case key in its sense of direction, with the upper case key working backward and/or up, while the lower case key moves forward and/or down.

Whenever you are typing in text--whether you begin with **i**, **a**, **o**, **O**, **s**, **C**, etc.--you can give many lines of input or just a few characters. To type in more than one line of text, hit a RETURN at the middle of your input. A new line is created for text, and you can continue to type. If you are on a slow and dumb terminal the editor may choose to wait to redraw the tail of the screen, and let you type over the existing screen lines. This avoids the lengthy delay that would occur if the editor attempted to keep the tail of the screen always up to date. The tail of the screen is fixed, and the missing lines reappear, when you hit ESC.

While you are inserting new text, you can use the characters you normally use at the system command level (usually ^**H** or #) to backspace over the last character you typed, and the character you use to kill input lines (usually @, ^**X**, or ^**U**) to erase the input you have typed on the current line. The character ^**W** erases a whole word and leaves you after the space after the previous word; it is useful for quickly backing up in an insert.

Notice that when you backspace during an insertion, the characters you backspace over are not erased; the cursor moves backwards, and the characters remain on the display. This is often useful if you are planning to type in something similar. In any case the characters disappear when when you hit ESC.

Notice also that you can't erase characters you didn't insert with the current insertion command, and that you can't backspace around the end of a line. If you need to back up to the previous line to make a correction, just hit ESC and move the cursor back to the previous line. After making the correction you can return to where you were and use the insert or append command again.

5.9.2 Making Small Corrections

You can make small corrections in existing text quite easily. Find a single character that is wrong or just pick any character. Use the arrow keys to find the character, or get near the character with the word motion keys and then either backspace (hit the BS key or ^**H** or even just **h**) or SPACE (using the space bar) until the cursor is on the character that is wrong. If the character is not needed then hit the **x** key; this deletes the character from the file. It is analogous to the way you **x** out characters when you make mistakes on a typewriter (except it's not as messy).

If the character is incorrect, you can replace it with the correct character by giving the command **r***c*, where *c* is replaced by the correct character. Finally if the character that is incorrect should be replaced by more than one character, give the command **s**, which substitutes a string of characters, ending with ESC. If a small number of characters are wrong, you can precede **s** with a count of the number of characters to be replaced. Counts are also useful with **x** to specify the number of characters to be deleted.

5.9.3 More Corrections: Operators

You already know almost enough to make changes at a higher level. All you need to know now is that the **d** key acts as a delete operator. Try the command **dw** to delete a word. Try hitting **.** a few times. Notice that this repeats the effect of the **dw**. The command **.** repeats the last command that made a change. You can remember it by analogy with an ellipsis '...'.

0. In fact, the character ^**H** (backspace) always works to erase the last input character here regardless of what your erase character is.

Now try **db**. This deletes a word backwards, namely the preceding word. Try **d**SPACE. This deletes a single character, and is equivalent to the **x** command.

Another very useful operator is **c** or change. The command **cw** thus changes the text of a single word. You follow it by the replacement text ending with an ESC. Find a word that you can change to another, and try this now. Notice that the end of the text to be changed was marked with the character '$' so that you can see this as you are typing in the new material.

5.9.4 Operating on Lines

You often want to operate on lines rather than just words or letters. Find a line you want to delete, and type **dd**, the **d** operator twice. This deletes the line. If you are on a dumb terminal, the editor may just erase the line on the screen, replacing it with a line with only an @ on it. This line does not correspond to any line in your file, but only acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen, which would be necessary to close up the hole created by the deletion on a terminal without a delete line capability.

Try repeating the **c** operator twice; this changes a whole line, erasing its previous contents and replacing them with text you type up to an ESC.

You can delete or change more than one line by preceding the **dd** or **cc** with a count, i.e. **5dd** deletes 5 lines. You can also give a command like **dL** to delete all the lines up to and including the last line on the screen, or **d3L** to delete through the third from the bottom line. Try some commands like this now.* Notice that the editor lets you know when you change a large number of lines so that you can see the extent of the change. The editor also always tells you when a change you make affects text you cannot see.

5.9.5 Undoing

Now suppose that the last change you made was incorrect; you could use the insert, delete and append commands to put the correct material back. However, since we often regret a change or make a change incorrectly, the editor provides a **u** (undo) command to reverse the last change you made. Try this a few times, and give it twice in a row to notice that an **u** also undoes a **u**.

0. The command **S** is a convenient synonym for for **cc**, by analogy with **s**. Think of **S** as a substitute on lines, while **s** is a substitute on characters.

0. * Using the / search after a **d** is subtle. This move normally deletes characters from the current position to the point of the match. If you want to delete whole lines including the two points, give the pattern as **/pat/+0**, a line address.

The undo command lets you reverse only a single change. After you make a number of changes to a line, you may decide that you would rather have the original state of the line back. The **U** command restores the current line to the state before you started changing it.

You can recover text that you delete, even if undo will not bring it back; see the section on recovering lost text below.

5.9.6 Summary

SPACE	advance the cursor one position
^H	backspace the cursor
^W	erase a word during an insert
erase	your erase (usually ^H or #), erases a character during an insert
kill	your kill (usually @, ^X, or ^U), kills the insert on this line
.	repeats the changing command
O	opens and inputs new lines, above the current
U	undoes the changes you made to the current line
a	appends text after the cursor
c	changes the object you specify to the following text
d	deletes the object you specify
i	inserts text before the cursor
o	opens and inputs new lines, below the current
u	undoes the last change

5.10 Moving about; Rearranging and Duplicating Text

5.10.1 Low Level Character Motions

Now move the cursor to a line containing a punctuation or a bracketing character such as a parenthesis or a comma or period. Try the command **f***x* where *x* is this character. This command finds the next *x* character to the right of the cursor in the current line. Try then hitting a ;, which finds the next instance of the same character. By using the **f** command and then a sequence of ;'s you can often get to a particular place in a line much faster than with a sequence of word motions or SPACEs. The **F** command, which is like **f**, searches backward. The ; command repeats **F** also.

When you are operating on the text in a line, you often find it useful to deal with the characters up to, but not including, the first instance of a character. Try **df***x* for some *x* now and notice that the *x* character is deleted. Undo this with **u** and then try **dt***x*; the **t** here stands for to, i.e. delete up to the next *x*, but not the *x*. The command **T** is the reverse of **t**.

When working with the text of a single line, an ↑ moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Hitting '0' (zero) also works to move to the beginning of a line. Thus **$a** will append new text at the end of the current line.

Your file may have tab (^I) characters in it. These characters are represented as a number of spaces expanding to a tab stop, where tab stops are every 8 positions.* When the cursor is at a tab, it sits on the last of the several spaces that represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed in the same way they are represented in this document, that is, with a two character code, the first character of which is '^'. On the screen non-printing characters resemble a '^' character adjacent to another, but if you space or backspace over the character, you see that the two characters are, like the spaces representing a tab character, a single character.

The editor sometimes won't allow you to insert control characters, depending on the character and the setting of the *beautify* option. You can force a control character into the file by beginning an insert and then typing a **^V** before the control character. The **^V** quotes the following character, causing it to be inserted directly into the file.

5.10.2 Higher Level Text Objects

Sometimes the capacity to work with characters or words or lines is not enough; you need to be able to manipulate sentences, paragraphs, and sections. A sentence is defined to end at a '.', '!' or '?', followed by either the end of a line, or by two spaces. Any number of closing ')', ']', '"' and ''' characters may appear after the '.', '!' or '?' before the spaces or end of line.

The operations **(** and **)** move to the beginning of the previous and next sentences respectively. Thus the command **d)** deletes the rest of the current sentence; likewise **d(** deletes the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

The operations { and } move over paragraphs and the operations **[[** and **]]** move over sections.

0. * This is settable by a command of the form **:se ts**=*x*CR, where *x* is the number of columns per tabstop. This has effect on the screen representation within the editor.

0. The **[[** and **]]** operations require the operation character to be doubled because they can move the cursor far from where it currently is. While it is easy to get back with the command ``, these commands would still be frustrating if they were easy to hit accidentally.

A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the definition of the string valued option *paragraphs*. The default setting for this option defines the paragraph macros of the *–ms* and *–mm* macro packages, i.e. the '.IP', '.LP', '.P' and '.QP', '.P' and '.LI' macros. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

Sections in the editor begin after each macro in the *sections* option, normally '.NH', '.SH', '.H' and '.HU', and each line with a formfeed **^L** in the first column. Section boundaries are always line and paragraph boundaries also.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different window size in which to redraw the screen at the new location, and this window size is the base size for newly drawn windows until another size is specified. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small window.

5.10.3 Rearranging and Duplicating Text

The editor has a single unnamed buffer where the last deleted or changed text is saved.

The operator **y** yanks a copy of the object that follows into the unnamed buffer. The text can then be put back in the file with the commands **p** and **P**; **p** puts the text after or below the cursor, while **P** puts the text before or above the cursor.

If the text that you yank forms a part of a line, or is an object such as a sentence, which partially spans more than one line, then when you put the text back, it is placed after the cursor (or before if you use **P**). If the yanked text forms whole lines, they are put back as whole lines, without changing the current line. In this case, the put acts much like a **o** or **O** command.

0. You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT. See section 6.2 for details. The '.bp' directive is also considered to start a paragraph.

Try the command **YP**. This makes a copy of the current line and leaves you on this copy, which is placed before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** also makes a copy of the current line, and places it after the current line. You can give **Y** a count of lines to yank, and thus duplicate several lines; try **3YP**.

To move text within the buffer, you need to delete it in one place, and put it back in another. An ordinary delete command saves the text in the unnamed buffer, so that an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you change files.

5.10.4 Summary

↑	first non-white on line
$	end of line
)	forward sentence
}	forward paragraph
]]	forward section
(	backward sentence
{	backward paragraph
[[	backward section
f*x*	find *x* forward in line
p	put text back, after cursor or below current line
y	yank operator, for copies and moves
t*x*	up to *x* forward, for operators
F*x*	f backward in line
P	put text back, before cursor or above current line
T*x*	t backward in line

5.11 High Level Commands

5.11.1 Writing, Quitting, Editing New Files

So far we have seen how to enter *vi* and to write out our file using either **ZZ** or **:w**CR. The first exits from the editor, (writing if changes were made), the second writes and stays in the editor.

If you have changed the editor's copy of the file but do not wish to save your changes, then you can give the command **:q!**CR to quit from the editor without writing the changes. You can also reedit the same file (starting over) by giving the command **:e!**CR. These commands should be used only rarely, and with caution, as you cannot recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving the editor by giving the command **:e** *name*CR. If you have not written out your file before you try to do this, then the editor will tell you this, and delay editing the other file. You can then give the command **:w**CR to save your work and then the **:e** *name*CR command again, or carefully give the command **:e!** *name*CR, which edits the other file discarding the changes you have made to the current file. To have the editor automatically save changes, include *set autowrite* in your EXINIT, and use **:n** instead of **:e**.

5.11.2 Escaping to a Shell

You can get to a shell to execute a single command by giving a *vi* command of the form :!*cmd*CR. The system runs the single command *cmd* and when the command finishes, the editor asks you to hit a RETURN to continue. When you have finished looking at the output on the screen, hit RETURN. The editor clears the screen and redraws it. You can then continue editing. You can also give another : command when it asks you for a RETURN; in this case the screen is not redrawn.

If you wish to execute more than one command in the shell, you can give the command **:sh**CR. This gives you a new shell, and when you finish with the shell, ending it by typing a **^D**, the editor clears the screen and continues.

On systems that support it, **^Z** suspends the editor and returns to the (top level) shell. When the editor is resumed, the screen is redrawn.

5.11.3 Marking and Returning

The command `` returned to the previous place after a motion of the cursor by a command such as /, **?** or **G**. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command **m***x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line (any way you like) and hit **`a**. The cursor returns to the place you marked. Marks last only until you edit another file.

Sometimes you mark a line in the middle but want to return to the beginning of the marked line; for example, when using operators such as **d** or **c** on marked lines. In this case you can use the form '*x* rather than `*x*. Used without an operator, '*x* will move to the first non-white character of the marked line; similarly '' moves to the first non-white character of the line containing the previous context mark ``.

5.11.4 Adjusting the Screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than the editor wrote output to your terminal, you can hit a **^L**, the ASCII form-feed character, to cause the screen to be refreshed.

On a dumb terminal, if there are @ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing **^R** to cause the editor to retype the screen, closing up these holes.

Finally, if you wish to place a certain line on the screen at the top middle or bottom of the screen, you can position the cursor to that line, and then give a **z** command. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a **.** if you want it at the center, or a – if you want it at the bottom.

5.12 Special Topics

5.12.1 Editing on Slow Terminals

When you are on a slow terminal, you will probably want to limit the amount of output generated to your screen so that you will not suffer long delays waiting for the screen to be refreshed. We have already pointed out how the editor optimizes the updating of the screen during insertions on dumb terminals to limit the delays, and how the editor erases lines to @ when they are deleted on dumb terminals.

The use of the slow terminal insertion mode is controlled by the *slowopen* option. You can force the editor to use this mode even on faster terminals by giving the command **:se slow**CR. ('Se' is short for 'set'.) If your system is sluggish this helps lessen the amount of output coming to your terminal. You can disable this option by **:se noslow**CR.

The editor can simulate an intelligent terminal on a dumb one. Try giving the command **:se redraw**CR. This simulation generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command

:se noredrawCR.

The editor also makes editing more pleasant at low speed by starting in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The editor can expand the window easily when you insert in the middle of the screen on these terminals. If possible, try the editor on an intelligent terminal to see how this works.

You can control the size of the window that is redrawn each time the screen is cleared by giving window sizes as argument to the commands that cause large screen motions:

: / ? [[ ]] ` '

Thus if you are searching for a particular instance of a common string in a file you can precede the first search command by a small number, say 3, and the editor draws three line windows around each instance of the string it locates.

You can easily expand or contract the window, placing the current line as you choose, by giving a number on a **z** command, after the **z** and before the following RETURN, **.** or **–**. Thus the command **z5.** redraws the screen with the current line in the center of a five line window.

If the editor is redrawing or otherwise updating large portions of the display, you can interrupt this updating by hitting a DEL or RUBOUT as usual. If you do this you may partially confuse the editor about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by hitting a ^**L**; or move or search again, ignoring the current state of the display.

See section 7.8 on *open* mode for another way to use the *vi* command **set** on slow terminals.

5.12.2 Options, Set, and Editor Startup Files

The editor has a set of options, some of which have been mentioned above. The most useful options are given in the following table.

Name	Default	Description
autoindent	noai	Supply indentation automatically
autowrite	noaw	Automatic write before **:n**, **:ta**, ^↑, **!**
ignorecase	noic	Ignore case in searching
lisp	nolisp	**(** { **)** } commands deal with S-expressions
list	nolist	Tabs print as ^I; end of lines marked with **$**
magic	nomagic	The characters . [and * are special in scans
number	nonu	Lines are displayed prefixed with line numbers
paragraphs	para=IPLPPPQPbpP LI	Macro names which start paragraphs
redraw	nore	Simulate a smart terminal on a dumb one

0. Note that the command **5z.** has an entirely different effect, placing line 5 in the center of a new window.

sections	sect=NHSHH HU	Macro names which start new sections
shiftwidth	sw=8	Shift distance for <, > and input ^**D** and ^**T**
showmatch	nosm	Show matching **(** or { as **)** or } is typed
slowopen	slow	Postpone display updates during inserts
term	dumb	The kind of terminal you are using.

The options are of three kinds: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form

set *opt=val*

and toggle options can be set or unset by statements of one of the forms

set *opt*
set no*opt*

These statements can be placed in your EXINIT in your environment, or given while you are running *vi* by preceding them with a : and following them with a CR.

You can get a list of all options that you have changed by the command **:set**CR; you can get the value of a single option by the command **:set** *opt***?**CR. A list of all possible options and their values is generated by **:set all**CR. Set can be abbreviated **se**. Multiple options can be placed on one line, e.g. **:se ai aw nu**CR.

Options set by the **set** command only last while you stay in the editor. You may want to have certain options set whenever you use the editor. This can be accomplished by creating a list of *ex* commands that are to be run every time you start up *ex*, *edit*, or *vi*. A typical list includes a series of **set** commands. Put these commands all on one line, and separate them with the | character, for example:

set ai aw terse

which sets the options *autoindent*, *autowrite*, and *terse*. This string should be placed in the variable EXINIT in your environment. If you use *csh*, put this line in the file *.login* in your home directory:

0. All commands that start with : are *ex* commands.

```
setenv EXINIT 'set ai aw terse
'
```

If you use the standard v7 shell, put these lines in the file *.profile* in your home directory:

```
EXINIT='set ai aw terse
'
export EXINIT
```

Of course, the particulars of the line depend on which options you want to set.

5.12.3 Recovering Lost Lines

You might have a serious problem if you deleted a number of lines inadvertently. Despair not, the editor saves the last 9 deleted blocks of text in a set of numbered registers 1–9. You can get the *n*'th previous deleted text back in your file by the command "*n***p**. The " here says that a buffer name is to follow, *n* is the number of the buffer you wish to inspect (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, hit **u** to undo this and then **.** (period) to repeat the put command. In general the **.** command repeats the last change you made, but in this case, when the last command refers to a numbered text buffer, the **.** command increments the number of the buffer before repeating the command. Thus a sequence of the form

```
"1pu.u.u.
```

if repeated long enough, shows you all the deleted text that has been saved for you. You can omit the **u** commands here to gather up all this text in the buffer, or stop after any **.** command to keep just the most currently recovered text. The command **P** can also be used rather than **p** to put the recovered text before rather than after the cursor.

5.12.4 Recovering Lost Files

If the system crashes, you can recover the work you were doing to within a few changes. Change to the directory you were in when the system crashed and give a command of the form:

```
% vi -r name
```

replacing *name* with the name of the file you were editing. This recovers your work to a point near where you left off.

You can get a listing of the files that are saved for you by giving the command:

```
% vi -r
```

The invocation "*vi -r*" does not always list all saved files, but they can be recovered with "vi -r *name*" even if they are not listed.

5.12.5 Continuous Text Input

When you are typing in large amounts of text it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command **:se wm=n**CR, where *n* is some number of columns. So, the command **:se wm=10**CR causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.

If the editor breaks an input line and you wish to put it back together you can tell it to join the lines with **J**. You can give **J** a count of the number of lines to be joined as in **3J** to join 3 lines. The editor supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. Kill the white space with **x** if you don't want it.

5.12.6 Features for Editing Programs

The editor has a number of commands for editing programs. Editing programs is different from editing text because in programs, an indentation structure must often be maintained. The editor has a *autoindent* facility to help you generate correctly indented programs.

To enable this facility you can give the command **:se ai**CR. Now try opening a new line with **o** and type some characters on the line after a few tabs. If you now start another line, notice that the editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **^D** key to backtab over the supplied indentation.

Each time you type **^D** you back up one position, normally to an 8 column boundary. This amount is settable; the editor has an option called *shiftwidth* that can be set to change this value. Try giving the command **:se sw=4**CR and then experimenting with autoindent again.

0. In rare cases, some of the lines of the file may be lost. The editor gives you the numbers of these lines and the text of the lines are replaced by the string 'LOST'. These lines almost always are among the last few you changed. You can either choose to discard the changes (if they are easy to remake) or replace the few lost lines by hand.

For shifting lines in the program left and right, the operators < and > shift the lines you specify right or left by one *shiftwidth.* Try << and >>, which shift one line left or right, and <**L** and >**L**, which shift the rest of the display left and right.

If you have a complicated expression and wish to see if the parentheses match, put the cursor at a left or right parenthesis and hit **%**. This will show you the matching parenthesis. This works also for braces { and }, and brackets [and].

If you are editing C programs, you can use the **[[** and **]]** keys to advance or retreat to a line starting with a {, i.e. a function declaration at a time. When **]]** is used with an operator it stops after a line that starts with }; this is sometimes useful with **y]]**.

5.12.7 Filtering Portions of the Buffer

You can run system commands over portions of the buffer using the operator **!**. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. Try typing in a list of random words, one per line and ending them with a blank line. Back up to the beginning of the list, and then give the command **!}sort**CR. This says to sort the next paragraph of material, and the blank line ends a paragraph.

5.12.8 Commands for Editing LISP

If you are editing a LISP program you should set the option *lisp* by doing **:se lisp**CR. This changes the **(** and **)** commands to move backward and forward over s-expressions. The { and } commands are like **(** and **)** but don't stop at atoms. These can be used to skip to the next list, or through a comment quickly.

The *autoindent* option works differently for LISP, supplying indent to align at the first argument to the last open list. If there is no such argument then the indent is two spaces more than the last level.

Another option useful for typing in LISP is *showmatch.* Try setting it with **:se sm**CR and then try typing a '(' some words and then a ')'. Notice that the cursor shows the position of the '(' which matches the ')' briefly. This happens only if the matching '(' is on the screen, and the cursor stays there for at most one second.

The editor also has an operator to realign existing lines as though they had been typed in with *lisp* and *autoindent* set. This is the = operator. Try the

0. The LISP features are not available on some v2 editors due to memory constraints.

command =% at the beginning of a function. This will realign all the lines of the function declaration.

When you are editing LISP, **[[** and **]]** advance and retreat to lines beginning with a **(**, and are useful for dealing with entire function definitions.

5.12.9 Macros

Vi has a parameterless macro facility, which lets you set it up so that when you hit a single keystroke, the editor will act as though you had hit some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two kinds of macros: Ones where you put the macro body in a buffer register, say *x*. You can then type @**x** to invoke the macro. The @ may be followed by another @ to repeat the last macro. You can use the *map* command from *vi* (typically in your *EXINIT*) with a command of the form:

:map *lhs rhs*CR

mapping *lhs* ('left hand side') into *rhs* ('right hand side'). Restrictions are: *lhs* should be one keystroke (either 1 character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish, and *vi* will wait for you to finish it before it echoes anything). The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab or newline into *lhs* or *rhs* you should escape them with a **^V**. (It may be necessary to double the **^V** if the map command is given inside *vi,* rather than in *ex.)* Spaces and tabs inside the *rhs* need not be escaped.

Thus to make the **q** key write and exit the editor, you can give the command

:map q :wq^V^VCR CR

which means that whenever you type **q**, the system will behave as though you had typed the four characters **:wq**CR. A **^V**'s is needed because without it the CR would end the : command, rather than becoming part of the *map* definition. Two **^V**'s are required because from within *vi*, two **^V**'s must be typed to get one. The first CR is part of the *rhs*, the second terminates the : command.

Macros can be deleted with

0. Plexus currently does not support the macro feature.

unmap lhs

If the *lhs* of a macro is "#0" through "#9", this maps the particular function key instead of the 2 character "#" sequence. So that terminals without function keys can access such definitions, the form "#x" will mean function key *x* on all terminals (and need not be typed within one second.) The character "#" can be changed by using a macro in the usual way:

```
:map ^V^V^I #
```

to use tab, for example. (This won't affect the *map* command, which still uses #, but just the invocation from visual mode.)

The undo command reverses an entire macro call as a unit, if it made any changes.

Placing a '!' after the word **map** causes the mapping to apply to input mode, rather than command mode. Thus, to arrange for **^T** to be the same as 4 spaces in input mode, you can type:

```
:map ^T ^Vbbbb
```

where **b** is a blank. The **^V** is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*. Word Abbreviations

A feature similar to macros in input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

```
:ab eecs Electrical Engineering and Computer Sciences
```

causes the word 'eecs' to always be changed into the phrase 'Electrical Engineering and Computer Sciences' Word abbreviation is different from macros in that only whole words are affected. If 'eecs' were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. An abbreviation need not be a single keystroke, as it should be with a macro.

5.12.10 Abbreviations

The editor has a number of short commands that abbreviate longer commands introduced here. You can find these commands easily on the quick reference card. They often save a bit of typing and you can learn them as convenient.

5.13 Nitty-gritty Details

5.13.1 Line Representation in the Display

The editor folds long logical lines onto many physical lines in the display. Commands that advance "lines" advance *logical* lines, not physical lines. A logical line is everything between user-input carriage returns. (So if you use autowrap and let the system break lines for you, the physical lines thus broken are not delimited by your carriage returns and thus do not correspond to *vi*'s idea of logical lines.)

The command **|** moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Try **80|** on a line which is more than 80 columns long.

The editor puts only full lines on the display; if there is not enough room on the display to fit a logical line, the editor leaves the physical line empty, placing only an @ on the line as a place holder. When you delete lines on a dumb terminal, the editor often just clears the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the **^R** command.

The editor can place line numbers before each line on the display. Give the command **:se nu**CR to enable this, and the command **:se nonu**CR to turn it off. However, this may not work well with long input lines on some dumb terminals. The display of long lines may be scrambled and the apparent cursor position may not be reliable; i.e., you may change something inadvertently. Be prepared to issue lots of **^L** commands.

You can have tabs represented as **^I** and the ends of lines indicated with '$' by giving the command **:se list**CR; **:se nolist**CR turns this off.

Finally, lines consisting of only the character ' ' are displayed when the last line in the file is in the middle of the screen. These represent physical lines that are past the logical end of file.

5.13.2 Counts

Most *vi* commands accept a preceding count to affect their behavior in some way. The following table gives the common ways in which the counts are used:

new window size	**: / ? [[ ]] ` ´**
scroll amount	**^D ^U**
line/column number	**z G |**
repeat effect	most of the rest

The editor maintains a notion of the current default window size. On terminals that run at speeds greater than 1200 baud, the editor uses the full terminal screen. On terminals slower than 1200 baud (most dialup lines are

0. You can make long lines very easily by using **J** to join together short lines.

in this group) the editor uses 8 lines as the default window size. At 1200 baud the default is 16 lines.

The editor uses this size when it clears and refills the screen after a search or other motion moves far from the edge of the current window. All the commands that take a new window size as count often cause the screen to be redrawn. If you anticipate this, but do not need as large a window as you are currently using, you may wish to change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen expands if you move off the top with a – or similar command or off the bottom with a command such as RETURN or ^**D**. The window reverts to the last specified size the next time it is cleared and refilled.

The scroll commands ^**D** and ^**U** likewise remember the amount of scroll last specified. Initially they use half the basic window size. The simple insert commands use a count to specify a repetition of the inserted text. Thus **10a**+——ESC inserts a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands that ignore any counts (such as ^**R**), the rest of the editor commands use a count to indicate a simple repetition of their effect. Thus **5w** advances five words on the current line, while **5**RETURN advances five lines. You can also give a count to the **.** command, which repeats the last changing command. If you do **dw** and then **3.**, you delete first one and then three words. You can then delete two more words with **2.**.

5.13.3 More File Manipulation Commands

The following table lists the file manipulation commands you can use when you are in *vi.*

0. But not by a ^**L**, which just redraws the screen as it is.

:w	write back changes
:wq	write and quit
:x	write (if necessary) and quit (same as ZZ).
:e *name*	edit file *name*
:e!	reedit, discarding changes
:e + *name*	edit, starting at end
:e +*n*	edit, starting at line *n*
:e #	edit alternate file
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:*x,y*w *name*	write lines *x* through *y* to *name*
:r *name*	read file *name* into buffer
:r !*cmd*	read output of *cmd* into buffer
:n	edit next file in argument list
:n!	edit next file, discarding changes to current
:n *args*	specify new argument list
:ta *tag*	edit file containing tag *tag*, at *tag*

All of these commands are followed by a CR or ESC. The most basic commands are **:w** and **:e**. A normal editing session on a single file ends with a **ZZ** command. If you are editing for a long period of time you can give **:w** commands occasionally after major amounts of editing, and then finish with a **ZZ**. When you edit more than one file, you can finish with one with a **:w** and start editing a new file by giving a **:e** command, or set *autowrite* and use **:n** <file>.

If you make changes to the editor's copy of a file, but do not wish to write them back, you must give an **!** after the command you would otherwise use; this forces the editor to discard any changes you have made. Use this carefully.

The **:e** command can be given a + argument to start at the end of the file, or a +*n* argument to start at line *n*. Actually, *n* may be any editor command not containing a space, usefully a scan like +/*pat* or +**?***pat*. In forming new names to the **e** command, you can use the character %, which is replaced by the current file name, or the character #, which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file name. Thus if you try to do a **:e** and get a diagnostic that you haven't written the file, you can give a **:w** command and then a **:e** # command to redo the previous **:e**.

You can write part of the buffer to a file by finding out the lines that bound the range to be written using ^**G**, and giving these numbers after the : and before the **w**, separated by ,'s (commas). For example, to write out lines 10 through 200 into a file called *pip*, issue the command

:10,200w pipCR

You can also mark these lines with **m** and then use an address of the form *´x,´y* on the **w** command here.

You can read another file into the buffer after the current line by using the **:r** command. You can also read in the output from a command; just use !*cmd* instead of a file name. So, for example, to read in the output of the *ls* command, type **:r !ls**.

If you wish to edit a set of files in succession, you can give all the names on the command line, and then edit each one in turn using the command **:n**. You can also respecify the list of files to be edited by giving the **:n** command a list of file names, or a pattern to be expanded as you would have given it in the initial *vi* command to the shell.

The **:ta** command is very useful for editing large programs. It utilizes a data base of function names and their locations--which can be created by programs such as *ctags*--to quickly find a function whose name you give. If the **:ta** command requires the editor to switch files, then you must :**w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again. (The tag feature is not available in some v2 editors.)

5.13.4 More about Searching for Strings

When you search for strings with / and **?**, the editor normally places you at the next or previous occurrence of the string. But sometimes you want to affect the lines *between* your current position and the next or previous occurrence of the pattern, without affecting the line containing the pattern. This is especially so if you are using an operator such as **d**, **c** or **y**. You can give a search of the form */pat/–n* to refer to the *n*'th line before the next line containing *pat*, or you can use + instead of – to refer to the lines after the one containing *pat*. If you don't give a line offset, then the editor just affects characters up to the match place, rather than whole lines; thus use "+0" to affect to the line that matches.

You can have the editor ignore the case of words in the searches it does by giving the command **:se ic**CR. The command **:se noic**CR turns this off.

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should

set nomagic

in your EXINIT. In this case, only the characters ↑ and **$** are special in patterns. The character \ is also then special (as it is most everywhere in the system), and may be used to get at the extended pattern matching facility. Thus, with *nomagic* set, the character \ functions to turn on magic;

with *magic* set, \ turns it off. You must also use a \ before a literal / in a forward scan or a **?** in a backward scan, in any case. The following table gives the extended forms when **magic** is set.

↑	at beginning of pattern, matches beginning of line
$	at end of pattern, matches end of line
.	matches any character
\<	matches the beginning of a word
\>	matches the end of a word
[*str*]	matches any single character in *str*
[↑*str*]	matches any single character not in *str*
[*x*–*y*]	matches any character between *x* and *y*
*	matches any number of the preceding pattern, including 0

If you use **nomagic** mode, then the **. [** and ***** primitives are given with a preceding \.

5.13.5 More about Input Mode

A number of characters can make corrections during input mode. These are summarized in the following table.

ˆH	deletes the last input character
ˆW	deletes the last input word, defined as by **b**
erase	your erase character, same as **ˆH**
kill	your kill character, deletes the input on this line
\	escapes a following **ˆH** and your erase and kill
ESC	ends an insertion
DEL	interrupts an insertion, terminating it abnormally
CR	starts a new line
ˆD	backtabs over *autoindent*
0ˆD	kills all the *autoindent*
↑**ˆD**	same as **0ˆD**, but restores indent next line
ˆV	quotes the next non-printing character into the file

The most usual way of making corrections during input mode is to type **ˆH** to correct a single character, or to type one or more **ˆW**'s to back over incorrect words. If you use # as your erase character in the normal system, it works like **ˆH**.

Your system kill character, normally @, **ˆX** or **ˆU**, erases all the input you have given on the current line. In general, you can neither erase input back around a line boundary nor erase characters that you did not insert with this insertion command. To make corrections on the previous line after a new line has been started you can hit ESC to end the insertion, move over and make the correction, and then return to where you were. The command **A**, which appends at the end of the current line, is often useful for continuing.

If you wish to type in your erase or kill character (say # or @) then you must precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a ^**V**. The ^**V** echoes as a ↑ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact you may type any character and it will be inserted into the file at that point.*

If you are using *autoindent* you can backtab over the indent that it supplies by typing a ^**D**. This backs up to a *shiftwidth* boundary. This works only immediately after the supplied *autoindent.*

When you are using *autoindent* you may wish to place a label at the left margin of a line. This is easy: type ↑ and then ^**D**. The editor moves the cursor to the left margin for one line, and restore the previous indent on the next. You can also type a **0** followed immediately by a ^**D** to kill all the indent and not have it come back on the next line.

5.13.6 Upper Case only Terminals

If your terminal has only upper case, you can still use *vi* by using the normal system convention for typing on such a terminal. Characters that you normally type are converted to lower case, and you can type upper case letters by preceding them with a \. The characters { } | ` are not available on such terminals, but you can escape them as \(\↑ \) \! \'. These characters are represented on the display in the same way they are typed.

5.13.7 Vi and Ex

Vi is actually one mode of editing within the editor *ex.* When you are running *vi* you can escape to the line oriented editor of *ex* by giving the command **Q**. All the : commands introduced above are available in *ex.* Likewise, most *ex* commands can be invoked from *vi* using :. Just give them without the : and follow them with a CR.

In rare instances, an internal error may occur in *vi.* In this case you get a diagnostic and are left in the command mode of *ex.* You can then save your work and quit, if you wish, by giving a command **x** after the *ex* prompt

0. * This is not quite true. The implementation of the editor does not allow the NULL (^@) character to appear in files. Also the LF (linefeed or ^**J**) character is used by the editor to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for the editor to echo the ↑ before you type the character. In fact, the editor treats a following letter as a request for the corresponding control character. This is the only way to type ^**S** or ^**Q**, since the system normally uses them to suspend and resume output and never gives them to the editor to process.

0. The \ character you give does not echo until you type another key.

:, or you can reenter *vi* by giving *ex* a *vi* command.

Some tasks are easier in *ex* than in *vi.* Systematic changes in line oriented material are particularly easy. You can read the advanced editing documents for the editor *ed* to find out more about this style of editing. Experienced users often mix their use of *ex* command mode and *vi* command mode to speed the work they are doing.

5.13.8 Open Mode: Vi on Hardcopy Terminals and "Glass TTYs"

If you are on a hardcopy terminal or a terminal that does not have a cursor that can move off the bottom line, you can still use the command set of *vi,* but in a different mode. When you give a *vi* command, the editor tells you that it is using *open* mode. This name comes from the *open* command in *ex,* which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is displayed.

In *open* mode the editor uses a single line window into the file, and moving backward and forward in the file causes new lines to be displayed, always below the current line. Two commands of *vi* work differently in *open:* **z** and **^R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the **^R** command retypes the current line. On such terminals, the editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types a number of \'s to show you the characters that are deleted. The editor also reprints the current line soon after such changes so that you can see what the line looks like again.

This mode is sometimes useful on very slow terminals that can support *vi* in the full screen mode. You can do this by entering *ex* and using an *open* command.

6. INTRODUCTION TO THE SHELL

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

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

The **shell** is both a command language and a programming language that provides an interface to the UNIX operating system. This volume describes, with examples, the UNIX operating system **shell**. The "Simple Commands" part of this section covers most of the everyday requirements of terminal users. Some familiarity with the UNIX operating system is an advantage when reading this section; refer to the section "BASICS FOR BEGINNERS". The "Shell Procedures" part of this section describes those features of the **shell** primarily intended for use within **shell** commands or procedures. These include the control-flow primitives and string-valued variables provided by the **shell**. A knowledge of a programming language would be helpful when reading this section. The last part, "Keyword Parameters", describes the more advanced features of the **shell**. See Table 5.A for a defined listing of grammar words used in this section.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX V Administrator Reference Manual*. Other references to entries of the form **name**(N), where "N" is a number (1 or 6) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX V User Reference Manual*. Entries where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX V Programmer Reference Manual*.

6.1 SIMPLE COMMANDS

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

```
who
```

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

```
ls -l
```

prints a list of files in the current directory. The argument *-l* tells **ls**(1) to print status information, size, and the creation date for each file.

6.1.1 Background Commands

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

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number following its creation. A list of currently active processes may be obtained using the **ps**(1) command.

6.1.2 Input/Output Redirection

Most commands produce output to the *standard output* that is initially connected to the terminal. This output may be directed to a file by the notation ">" thus:

```
ls -l >file
```

The notation *>file* is interpreted by the **shell** and is not passed as an argument to **ls**(1). If *file* does not exist, the **shell** creates it; otherwise, the original contents of *file* are replaced with the output from **ls**(1). Output may be appended to a file using the notation ">>" as follows:

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

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

The *standard input* of a command may be taken from a file instead of the terminal by the notation "<" thus:

```
wc <file
```

The command **wc**(1) reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc -l <file
```

can be used.

6.1.3 Pipelines and Filters

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

```
ls -l | wc
```

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

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

except that no *file* is used. Instead the two processes are connected by a pipe [see **pipe**(2)] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc**(1) when there is nothing to read and halting **ls**(1) when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1) selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string "old". Another useful filter is **sort**(1). For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

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

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

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

6.1.4 File Name Generation

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

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

prints only information relating to the file *main.c*. The "**ls -l**" command alone prints the same information about all files in the current directory.

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

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

generates as arguments to **ls**(1) all file names in the current directory that end in *.c*. The character "*" is a pattern that will match any string including the null string. In general, *patterns* are specified as follows:

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

For example,

```
[a−z]*
```

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

The input

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

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

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

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

finds and prints the names of all *core* files in subdirectories of */usr/fred*. [The **echo**(1) command is a standard UNIX operating system command that prints its arguments, separated by blanks.] This last feature can be expensive, requiring a scan of all subdirectories of */usr/fred*.

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

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".". The input

```
echo .*
```

will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory", respectively. [Notice that **ls**(1) suppresses information for the files "." and "..".]

6.1.5 Quoting

Characters that have a special meaning to the **shell**, such as

< > * ? | &

are called *metacharacters*. A complete list of metacharacters is given in Table 5.B. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single **?**, and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line, the sequence **new line** (or RETURN) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

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

will echo

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

The quoted string may not contain a single quote but may contain new lines which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters. Details of quoting are described under "Evaluation and Quoting" in part "Keyword Parameters".

6.1.6 Prompting by the Shell

When the **shell** is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is "**$** ". The prompt may be changed by entering

```
PS1=newprompt
```

This sets the prompt to be the string "newprompt". If a new line is typed and further input is needed, the **shell** will issue the prompt "> ". Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (DEL) will return the **shell** to read another command. The other prompt (">") may be changed by entering:

```
PS2=more
```

6.1.7 The Shell and Login

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

6.1.8 Summary

ls Prints the names of files in the current directory.

ls >file Puts the output from **ls** into *file.*

ls | wc −l Prints the number of files in the current directory.

ls | grep old Prints those file names containing the string "old".

ls | grep old | wc −l Prints the number of files whose name contains the string "old".

cc pgm.c & Runs **cc** in the background.

6.2 SHELL PROCEDURES

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

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

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

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All UNIX operating system files have three independent attributes (often called "permissions"), *read*, *write*, and *execute* (rwx). The UNIX operating system command **chmod**(1) may be used to make a file executable. For example,

```
chmod +x wg
```

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

```
          wg fred
```

is equivalent to the call

```
        sh wg fred
```

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

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

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

```
nroff −T450 −cm $*
```

which simply prepends some arguments to those already given.

6.2.1 Control Flow—for

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

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

The text of *tel* is

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

The command

```
tel fred
```

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

The command

```
tel fred bert
```

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

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

```
for name in w1 w2
do
    command-list
done
```

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

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

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

The command

```
create alpha beta
```

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

6.2.2 Control Flow—case

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

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

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
append file
```

$# is the string "1", and the standard input is appended (copied) onto the end of *file* using the **cat**(1) command.

```
append file1 file2
```

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

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

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

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

Caution: **No check is made to ensure that only one pattern matches the case argument.**

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

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

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command.

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

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

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

is equivalent to

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

The usual quoting conventions apply so that

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

will match the character **?**.

6.2.3 Here Documents

The **shell** procedure *tel* described under "A. Control Flow—for" in this section uses the file */usr/lib/telnos* to supply the data for **grep(1)**. An alternative is to include this data within the **shell** procedure as a *here* document, as in,

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

In this example, the **shell** takes the lines between <<! and ! as the standard input for **grep**(1). The string "!" is arbitrary. The document is being terminated by a line that consists of the string following <<.

Parameters are substituted in the document before it is made available to **grep**(1) as illustrated by the following procedure called *edg*.

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

The call

```
edg string1 string2 file
```

is then equivalent to the command

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

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

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

[This version of *edg* is equivalent to the first except that **ed**(1) will print a **?** if there are no occurrences of the string **$1**.]

Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

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

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

6.2.4 Shell Variables

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

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

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

```
null=
```

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

```
echo $user
```

will echo *fred*.

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

For example,

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

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

```
echo ${user}
```

which is equivalent to

```
echo $user
```

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

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

will direct the output of **ps**(1) to the file */tmp/psa,* whereas,

```
ps a >$tmpa
```

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

Except for **$?**, the following are set initially by the **shell**.

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

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

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

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

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

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

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

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

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

$HOME	The default argument for the **cd**(1) command. The current directory is used to resolve file name references that do not begin with a / and is changed using the **cd** command.

For example,

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

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

```
        cat wn
```

will print on the terminal the file *wn* in this directory. The command **cd**(1) with no argument is equivalent to

```
        cd $HOME
```

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

$PATH	A list of directories containing commands (the *search path*). Each time a command is executed by the **shell**, a list of directories is searched for an executable file. If *$PATH* is not set, the current directory, */bin*, and */usr/bin* are searched by default. Otherwise, *$PATH* consists of directory names separated by :. For example,

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

specifies that the current directory (the null string before the first :), */usr/fred/bin, /bin,* and */usr/bin* are to be searched in that order. In this way, individual users can have their own 'private' commands that are accessible independently of the current directory. If

	the command name contains a /, this directory search is not used; a single attempt is made to execute the command.
$PS1	The primary **shell** prompt string, by default, "**$** ".
$PS2	The **shell** prompt when further input is needed, by default, "> ".
$IFS	The set of characters used by *blank interpretation*. (See "D. Evaluation and Quoting" in part "Keyword Parameters".)

6.2.5 Test Command

The **test** command is intended for use by **shell** programs. For example,

```
test −f file
```

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

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

6.2.6 Control Flow—while

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

A **while** loop has the general form

```
while command-list1
do
    command-list2
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed;

otherwise, the loop terminates. For example,

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

is equivalent to

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

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

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

```
until test -f file
do
    sleep 300
done
commands
```

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

6.2.7 Control Flow—if

Also available is a general conditional branch of the form,

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

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

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

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

An example of the use of **if, case,** and **for** constructions is given in "I. The Man Command" in part "Shell Procedures".

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

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

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

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

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

The following example is the **touch** command:

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

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

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

make a link to the file and then remove it.

The sequence

```
if command1
then
    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

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

Command Grouping

Commands may be grouped in two ways,

{ *command-list* ; }

and

(*command-list*)

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

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

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

The commands

```
cd x; rm junk
```

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

6.2.8 Debugging Shell Procedures

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

```
set −v
```

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

```
sh −v proc ...
```

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

The command

```
set −x
```

will produce an execution trace with flag **−x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

```
set −
```

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

6.2.9 The "man" Command

The following discussion of the man command assumes the existence of the document preparation features available as an option on the UNIX system.

The following is the **man** command which is used to print sections of the *Sys5 UNIX V User Reference Manual*. It is called by entering

```
man sh
man −t ed
man 2 fork
```

In the first call, the manual section for **sh** is printed. Since no section is specified, section 1 is used. The second call will typeset (**−t** option) the manual section for **ed**. The last call prints the **fork** manual page from section 2 of the manual.

6.3 KEYWORD PARAMETERS

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

```
user=fred command
```

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

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

For example,

```
set − *
```

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

6.3.1 Parameter Transmission

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

```
export user box
```

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

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

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

6.3.2 Parameter Substitution

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

```
echo $d
```

or

```
echo ${d}
```

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

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

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

```
echo ${d- ´*´}
```

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

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

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

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

which substitutes the same string as

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

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

If there is no sensible default, the notation

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

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

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

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

6.3.3 Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command **pwd**(1) prints on its standard output the name of the current directory. For example, if the current directory is */usr/fred/bin,* the command

```
d=‘pwd’
```

is equivalent to

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

The entire string between single quotes (‘...’) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ‘ must be escaped using a \.

For example,

```
ls ‘echo "$1"’
```

is equivalent to

```
ls $1
```

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

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

will print the string “main”. Its use is illustrated by the following fragment from a **cc**(1) command.

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

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

Here are some composite examples.

- for i in 'ls −t'; do ...

The variable *i* is set
to the names of files in time order,
most recent first.

- set 'date'; echo $6 $2 $3, $4

will print, e.g.,
1977 Nov 1, 23:59:59

6.3.4 Evaluation and Quoting

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

Commands are parsed initially according to the grammar given in Table 5.A. Before a command is executed, the following substitutions occur:

1. Parameter substitution, e.g., **$user**
2. Command substitution, e.g., **'pwd'**

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

   ```
   echo $X
   ```

   will echo "$y".
3. Blank interpretation

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

   ```
   echo ´ ´
   ```

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

```
echo $null
```

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

4. File name generation

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

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

As well as the quoting mechanisms described earlier using \ and ´...´, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not.

The following characters have a special meaning within double quotes and may be quoted using \.

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

For example,

```
echo "$x"
```

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

```
echo "$*"
```

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

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

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

```
echo "$@"
```

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

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

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

metacharacter

	\	$	*	`	"	´
´	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t	=	terminator
y	=	interpreted
n	=	not interpreted

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

```
eval echo $X
```

will echo the string "pqr".

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

```
wg=´eval who | grep´
$wg fred
```

is equivalent to

```
who | grep fred
```

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

6.3.5 Error Handling

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

Execution of a command (see also "G. Command Execution") may fail for any of the following reasons:

- Input/output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault" signal.

- The command terminates normally but returns a nonzero exit status.

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

- Syntax errors, e.g., if ...then... done
- A signal such as interrupt. The **shell** waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as **cd**(1).

The **shell** flag **−e** causes the **shell** to terminate if any error is detected. The following is a list of the UNIX operating system signals:

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

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

6.3.6 Fault Handling

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

```
trap ´rm /tmp/ps$$; exit´ 2
```

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

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

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

UNIX operating system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.
3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the **shell** procedure, for example, by invoking it in the background (see "G. Command Execution"), **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command illustrated below:

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

The cleanup action is to remove the file *junk$$*. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

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

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following:

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

is a fragment taken from the **nohup**(1) command which causes the UNIX operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
trap 2 3
```

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

```
trap
```

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

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

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

6.3.7 Command Execution

To run a command (other than a built-in), the **shell** first creates a new process using the system call **fork**(2). The execution environment for the command includes input, output, and the states of signals and is established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no fork is required and simply replaces the **shell** with a new command. For example, a simple version of the **nohup** command looks like

```
trap ´´ 1 2 3 15
exec $*
```

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

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

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

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

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

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

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

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

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

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

The environment for a command run in the background such as

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

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the **shell** and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

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

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

6.3.8 Invoking the Shell

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

−**c** *string*	If the −**c** flag is present, then commands are read from *string*.
−**s**	If the −**s** flag is present or if no arguments remain, commands are read from the standard input. **Shell** output is written to file descriptor 2.
−**i**	If the −**i** flag is present or if the **shell** input and output are attached to a terminal [as told by **getty**(8)], this **shell** is *interactive*. In this case, TERMINATE is ignored (so that **kill 0** does not kill an interactive **shell**, and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the **shell**.

TABLE 5.A

GRAMMAR

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

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

pipeline:	*command* *pipeline* | *command*
andor:	*pipeline* *andor* **&&** *pipeline* *andor* || *pipeline*
command-list:	*andor* *command-list* ; *command-list* **&** *command-list* ; *andor* *command-list* **&** *andor*
input-output:	> *word* < *word* >> *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:	sequence of nonblank characters
name	sequence of letters, digits, or underscores starting with a letter
digit:	**0 1 2 3 4 5 6 7 8 9**

TABLE 5.B

METACHARACTERS AND RESERVED WORDS

(a) *syntactic:*

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

(b) *patterns:*

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

(c) *substitution:*

${...}	substitute **shell** variable
'...'	substitute command output

(d) *quoting:*

\	quote the next character
´...´	quote the enclosed characters except for the ´
"..."	quote the enclosed characters except for the **$**, ' ,\, and "

(e) *reserved words:*

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

7. csh

7.1 The Basic Notion of Commands

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

Commands in the UNIX system expect a list of strings or *words* as arguments. Thus the command

```
mail bill
```

consists of two words. The first word, 'mail', names the command to be executed, in this case the *mail* program, which sends messages to other users. The shell uses the name of the command in attempting to run it. It looks in a number of *directories* for a file with the name *mail* and expects the file called "mail" to contain the *mail* program.

The rest of the words of the command are given to the command itself to execute. In this case the word *bill* is also specified; this is interpreted by the *mail* program to be the name of a user to whom mail is to be sent.

For example, Chris can send mail to Bill as follows.

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

Here Chris typed a message to send to *bill* and ended this message with a control-d, which sent an end-of-file to the *mail* program. The *mail* program then transmitted the message. The characters '% ' (per cent sign followed by space) were printed before and after the *mail* command by the shell to indicate that the shell was awaiting input.

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

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

7.2 Flag Arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names, some arguments specify optional capabilities of commands. By convention, such arguments begin with the character '–'. Thus the command

```
ls
```

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

```
ls –s
```

causes *ls* to also give, for each file, the size of the file in blocks of 1024 characters. The manual page for each command in the *Plexus Sys5 UNIX Programmer's Reference Manual* gives the available options for each command. The *ls* command has many useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands that are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard-to-remember options.

7.3 Output to Files

The concepts of *standard input* and *standard output* are very important in UNIX. The default for both is the terminal; this means that unless you tell UNIX otherwise, UNIX expects to receive input for its commands from the terminal and send output of commands to the terminal. But often you want to read input from or write output to *files* rather than simply taking input and output from the terminal. The shell provides simple ways to accomplish this.

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

```
date
```

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

```
date > now
```

runs the *date* command such that its standard output is the file 'now' rather than our terminal. Thus this command places the current date and time in the file 'now'. Note that the *date* command is unaware that its output is going to a file rather than to our terminal. *Date* sends its results to standard output, however that standard output is currently defined--terminal or file or other device or whatever. The *shell* performed this *redirection* before the command began executing.

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

7.4 Metacharacters in the Shell

The shell has a large number of special characters (like '>') that indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters that are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation*, which allows us to create words that contain *metacharacters* and to thus work without constantly worrying about whether certain characters are metacharacters.

Note that the C-shell is only reading input when it has prompted with '% '. Thus metacharacters normally have effect only then. So, for example, we need not worry about placing shell metacharacters in a letter we are sending via *mail.*

7.5 Input from Files; Pipelines

We learned above how to route the standard output of a command to a file. We can also route the standard input of a command from a file. This is not often necessary, however, since most commands will read from a file name given as argument.

For example, we can give the command

```
sort < data
```

to run the *sort* command with standard input from the file 'data'. But we would more likely say

```
sort data
```

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

We should note that if we just typed

sort

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

We can even combine the standard output of one command with the standard input of the next, i.e. to run the commands in a sequence known as a *pipeline.* This is an extremely useful feature. For instance, the command

ls –s

normally produces a list of the files in our directory with the size, in 1024-byte blocks, of each. If we are interested in learning which of our files is largest, we may wish to have this list sorted by size rather than by name--*ls* by default sorts by name. We could investigate the many options of *ls* to see if one lists in order of size, but we would eventually discover that no such option exists. Instead the shell lets us use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

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

ls –s | sort –n

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

ls –s | sort –n –r | head –5

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines of *its* standard input. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

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

7.6 Filenames

Many commands need the names of files as arguments. UNIX pathnames consist of a number of components separated by '/'. Each component except the last names a directory in which the next component resides. Thus the pathname

/etc/motd

specifies a file in the directory 'etc', which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd', which stands for 'message of the day'. Filenames that do not begin with '/' are interpreted starting at the current *working* directory. This directory is, by default, your *home* directory and can be changed dynamically by the *chdir* or *cd* change directory command.

Most filenames consist of a number of alphanumeric characters and '.'s. In fact, all printing characters except '/' may appear in filenames. However, non-alphabetic charcters usually do not belong in filenames, because many of these have special meaning to the shell. The character '.' is not a shell-metacharacter and is often used as the prefix with an *extension* of a *basename.* Thus

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

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

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

prog.*

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

echo prog.*

echoes the names

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

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

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

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

echoes (i.e., writes on standard output) a line of filenames; first those with one-character names, then those with two-character names, and finally those with three-character names. The names of each length will be independently lexicographically sorted.

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

```
prog.[co]
```

will match

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

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

```
chap.[1–5]
```

might match files

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

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

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

```
No match.
```

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

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

A special case of this notation consists of a ‘ ’ alone, e.g. ‘ /mbox’. This notation is expanded by the shell into the file ‘mbox’ in your *home* directory. This can be very useful if you have used *chdir* to change to another user’s directory and have found a file you wish to copy using *cp*. You can do

```
cp thatfile
```

which will be expanded by the shell to

```
cp thatfile /mnt/bill
```

i.e., the copy command interprets this as a request to make a copy of ‘thatfile’ in the directory ‘/mnt/bill’. The ‘ ’ notation doesn’t, by itself, force named files to exist. This is useful, for example, when using the *cp* command, e.g.

```
cp thatfile  /saveit
```

A mechanism using the characters ‘{’ and ‘}’ abbreviates a set of words that have common parts but cannot be abbreviated by the above mechanisms because they are not files, or are the names of files that do not yet exist. This mechanism will be described much later, in section 4.2, as it is used much less frequently.

7.7 Quotation

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

```
echo *
```

does not echo the character ‘*’. It either echoes a sorted list of filenames in the current directory, or prints the message ‘No match’ if there are no files in the current directory.

The recommended mechanism for placing characters that are neither numbers, digits, ‘/’, ‘.’ or ‘–’ in an argument word to a command is to enclose it with single quotation characters ‘´’, e.g.

echo ′*′

The *history* mechanism of the shell uses the special character '!', so '!' cannot be *escaped* in this way. It and the character '′' itself can be preceded by a single '\' to prevent their special meaning. These two mechanisms suffice to place any printing character into a word that is an argument to a shell command.

7.8 Terminating Commands

When you are running a command from the shell and the shell is waiting for it to complete, there are a couple of ways in which you can force such a command to complete. For instance if you type the command

cat /usr/man/docs/csh/csh

the system prints this document on your terminal. This will continue for several minutes unless you stop it. You can send an INTERRUPT signal to the *cat* command by hitting the DEL or RUBOUT key on your terminal. Actually, hitting this key sends this INTERRUPT signal to all programs running on your terminal, including your shell. The shell normally ignores such signals, however, so that the only program affected by the INTERRUPT is *cat.* Since *cat* does not take any precautions to catch this signal the INTERRUPT causes it to terminate. The shell notices that *cat* has died and prompts you again with '% '. If you hit INTERRUPT again, the shell just repeats its prompt, since it catches INTERRUPT signals but continues to execute commands anyway. If the shell went away like *cat,* this would log you out.

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

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

mail bill < prepared.text

the *mail* command terminates without our typing a control-d. This is because it read to the end-of-file of our file 'prepared.text'. We could also have done

cat prepared.text | mail bill

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

If you write or run programs that are not fully debugged, then you may want to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, generated by a control-\. This usually provokes the shell to produce a message like:

a.out: Quit — Core dumped

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

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

7.9 What Now?

We have so far seen a number of mechanisms of the shell and learned something about the way in which it operates. The remaining sections will go further into the internals of the shell, but you will surely want to try using the shell before you go any further. To get the C shell as your login shell, your entry in the file */etc/passwd* must be modified. */etc/passwd* is writable only by the superuser, so you may have to get your system administrator to perform the change. The last field of each line in */etc/passwd* is the shell field. No entry in this field means this user has the regular shell, */bin/sh*. This field of your line must be modified to read */usr/plx/csh*. You will get the C shell the next time you log in. It's a good idea to have your .login and .cshrc files set up appropriately before you try the C shell, so your terminal will behave properly.

You can also invoke a csh by typing '/usr/plx/csh'. This gives you a temporary C shell, and you may return to your login shell by typing control-d.

Much of the discussion in this manual so far is applicable to '/bin/sh' as well as the C shell. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

7.10 Details on the Shell

7.10.1 Shell Startup and Termination

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

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

```
tset –d adm3a
set history=20
set time=3
```

This file contains four commands to be executed by UNIX each time the user logs in. The first is a *tset* command, which informs the system that this user usually dials in on a Lear-Siegler ADM–3A terminal. The next two *set* commands are interpreted directly by the shell and affect the values of certain shell variables to modify the future behavior of the shell. Setting the variable *time* tells the shell to print time statistics on commands that take more than a certain threshold of machine time (in this case 3 CPU seconds). Setting the variable *history* tells the shell how much history of previous command words it should save in case the user wishes to repeat or rerun modified versions of previous commands. This mechanism involves a certain overhead, so the shell does not set this variable by default. The value of 20 is a reasonably large value to assign to *history.* More casual users of the *history* mechanism would probably set a value of 5 or 10. The use of the *history* mechanism will be described subsequently.

After executing commands from *.login,* the shell reads commands from your terminal, prompting for each with '% '. When it receives an end-of-file from the terminal, the shell prints 'logout' and executes commands from the file '.logout' in your home directory. After that the shell dies and UNIX logs you off the system. If the system is not going down, you receive a new login message. In any case, after the 'logout' message, the shell from which the end-of-file was received is doomed and takes no further input from the terminal.

7.10.2 Shell Variables

The shell maintains a set of *variables.* We saw above the variables *history* and *time,* which had values '20' and '3'. In fact, each shell variable has as value an array of zero or more *strings.* Shell variables may be assigned values by the *set* command. This command has several forms, the most useful of which was given above and is

```
set name=value
```

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

One of the most important variables is the variable *path.* This variable contains a sequence of directory names where the shell searches for commands. The *set* command shows the value of all variables currently defined (we usually say *set)* in the shell. The default value for path will be shown by *set* to be

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

This notation indicates that the variable *path* points to the current directory '.' and then '/bin' and '/usr/bin'. Commands that you may write might be in '.' (usually one of your directories). The most heavily used system commands live in '/bin'. Less heavily used system commands live in '/usr/bin'.

A number of useful programs that are not part of standard UNIX SYSTEM III--including *csh*--live in the directory '/usr/plx'. If you want all shells that you invoke to have access to these new programs, place the command

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

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

```
set
```

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

Other useful built-in variables are the variable *home,* which shows your home directory, and the variable *ignoreeof,* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal. To logout from UNIX with *ignoreeof* set you must type

logout

This is one of several variables that the shell does not care about the value of, only whether they are *set* or *unset.* Thus to set this variable you simply do

set ignoreeof

and to unset it do

unset ignoreeof

Both *set* and *unset* are built-in commands of the shell.

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

> **filename**

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

set noclobber

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

date > now

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

date >! now

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

If you receive mail frequently while you are logged in, and wish to be informed of the arrival of this mail, you can put a command

set mail=/usr/mail/yourname

in your *.login* file. Here you should change 'yourname' to your login name. The shell looks at this file every 10 minutes to see if new mail has arrived. If you receive mail only infrequently, you are better off not setting this variable; it only delays the shell's response to you.

The use of shell variables to introduce text into commands, which is most useful in shell command scripts, will be introduced in section 2.4.

7.10.3 The Shell's History List

The shell can maintain a history list into which it places the words of previous commands. You can use a metanotation to reintroduce commands or words from commands in forming new commands. This mechanism can repeat previous commands or correct minor typing mistakes in commands.

Consider the following transcript:

```
% ls -l shell.proc
-rw-r--r-- 1 sandy   6506 Jan 19 20:05 shell.proc
% chmod 755 !$
chmod 755 shell.proc
```

Here we asked for a long (-l) listing of the file 'shell.proc' and were told, among other things, that it was not executable. We need to change the permissions attached to it, so we execute a *chmod* command on '!$'. '!$' is a history notation that means the last word of the last command executed, in this case 'shell.proc'. The shell performed this substitution and then echoed the command ('chmod 755 shell.proc') as it would execute it. Suppose now that we want to verify that the permission changes were made by doing another 'ls -l'. We can do

```
% !l
-rwxrwxrwx 1 sandy   6506 Jan 19 20:05 shell.proc
%
```

We repeat the *ls*-l command with the history notation '!l', which repeats the last command that began with a word of which 'l' is a prefix.

The form '!!' re-executes the last command. Another useful command form is '↑lhs↑rhs', which performs a substitution similar to that in *ed* or *ex.* Thus in this example,

```
% cat  bill/csh/sh..c
/mnt/bill/csh/sh..c: No such file or directory
% ↑..↑.
cat  bill/csh/sh.c
#include "sh.h"

char    *pathlist[] =    { SRCHP
%
```

we used the substitution to correct a typing mistake, and then rubbed the command out after we saw that we had found the file we wanted. The substitution changed the two '.' characters to a single '.' character.

After this command we might do

```
% !! | lpr
cat  bill/csh/sh.c | lpr
```

to put a copy of this file on the line printer, or (immediately after the *cat* that worked above)

```
% pr !$ | lpr
pr  bill/csh/sh.c | lpr
%
```

More advanced forms of the history mechanism also exist. Substitutions themselves may be modified, so you can say (after the first successful *cat* above).

```
% cd !$:h
cd  bill/csh
%
```

The trailing ':h' on the history substitution here causes only the head portion of the pathname reintroduced by the history mechanism to be substituted. This and related mechanisms are used less often than the forms above.

A complete description of history mechanism features is given in the C shell manual entry in the *Plexus Sys5 UNIX Programmer's Reference Manual.*

7.10.4 Aliases

The shell has an *alias* mechanism that can simplify the commands you type, supply default arguments to commands, or perform transformations on commands and their arguments. The alias facility is similar to the macro facility of many assemblers.

Some of the features obtained by aliasing can also be obtained using shell command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment. Thus commands such as *chdir,* which must be done in the current shell, may not work the way you expect.

As an example, suppose you wish to use a new version of the *mail* program. The new program is called 'Mail', and the standard mail program continues to be called 'mail'. If you place the shell command

```
alias mail Mail
```

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

```
mail bill
```

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

```
alias ls ls –s
```

or even

```
alias dir ls –s
```

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

```
        dir  bill
```

then the shell translates this to

```
        ls -s /mnt/bill
```

Thus the *alias* mechanism can provide deafult arguments and short names for commands, and can define new short commands in terms of other commands. You can also 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 ´
```

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

```
        alias whois ´grep \!↑ /etc/passwd´
```

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

7.10.5 Detached Commands; >> and >& Redirection

A few more metanotations are useful. The metacharacter '&' may be placed after a command, or after a sequence of commands separated by ';' or '|'. This causes the shell not to wait for the commands to terminate before prompting again. We say that they are *detached* or *background* processes. Thus

```
        % pr  bill/csh/sh.c | lpr &
        5120
        5121
        %
```

Here the shell printed two numbers and came back very quickly rather than waiting for the *pr* and *lpr* commands to finish. These numbers are the process numbers assigned by the system to the *pr* and *lpr* commands.†

†Running commands in the background like this tends to slow down the system and is not a good idea if the system is overloaded. When overloaded, the system will just bog down more if you run a large number of processes at once.

Since havoc would result if a command run in the background were to read from your terminal at the same time as the shell does, the default standard input for a command run in the background is not your terminal, but an empty file called '/dev/null'. Commands run in the background are also made immune to INTERRUPT and QUIT signals that you may subsequently generate at your terminal.*

> ***If a background command stops suddenly when you hit INTERRUPT or QUIT it is likely a bug in the background program.**

If you intend to log off the system before the command completes you must run the command immune to HANGUP signals. This is done by placing the word 'nohup' before each program in the command, i.e.:

```
nohup man csh | nohup lpr &
```

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 directed to a file or a pipe. You may occasionally want to direct the diagnostic output along with the standard output. For instance, you may want to redirect the output of a long-running command into a file and have a record of any error diagnostic it produces. The command

```
command >& file
```

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

```
command |& lpr
```

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

#A command form

```
command >&! file
```

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

Finally, you can use the form

#A command form
command >> file

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

†If

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

```
command >>! file
```

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

7.10.6 Useful Built-in Commands

We now describe a few of the useful built-in commands of the shell.

The *alias* command described above assigns new aliases and shows existing aliases. With no arguments it prints the current aliases. It may also be given an argument such as

```
alias ls
```

to show the current alias for, in this case, 'ls'.

The *cd* and *chdir* commands are equivalent; they change the working directory of the shell. An experienced UNIX user usually makes a subdirectory for each of his projects and places all files related to each project in the appropriate subdirectory. Thus after you login you can do

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

after which you will be in the directory *newpaper.* You can return to your 'home' login directory by doing just

```
chdir
```

with no arguments. We used the *pwd* (print working directory) command to show the name of the current directory. The current directory is usually a subdirectory of your home directory. Thus, the home directory (here '/mnt/bill') path forms the prefix of the new directory name. In the example above, '/mnt/bill' forms the prefix of '/mnt/bill/newpaper'.

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

The *history* command shows the contents of the history list. The numbers given with the history events can be used to refer to previous events that are difficult to refer to using the contextual mechanisms introduced above. There is also a shell variable called *prompt.* By placing a '!' character in its value the shell there substitutes the index of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
        set prompt= \! % ´
```

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

The *logout* command terminates a login shell that has *ignoreeof* set.

The *repeat* command repeats a command. Thus to make 5 copies of the file *one* in the file *five* you could do

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

The *setenv* command sets variables in the environment. Thus

```
        setenv TERM adm3a
```

sets the value of the environment variable TERM to 'adm3a'. A user program *printenv* prints out the environment. It might then show:

```
        % printenv
        HOME  /usr/bill
        SHELL /bin/csh
        TERM  adm3a
        %
```

The *source* command forces the current shell to read commands from a file. Thus

```
        source .cshrc
```

causes any changes to *.cshrc* to take effect before the next time you login.

The *time* command can time a command no matter how much CPU time it takes. Thus

```
        % time cp five five.save
        0.0u 0.3s 0:01 26%
        % time wc five.save
          1200    6300   37650 five.save
        1.2u 0.5s 0:03 55%
        %
```

indicates that the *cp* command used less that 1/10th of a second of user time and only 3/10th of a second of system time in copying the file 'five' to 'five.save'. The command word count *wc,* which counts the number of words, character and lines in 'five.save', used 1.2 seconds of user time and 0.5 seconds of system time in 3 seconds of elapsed time. The percentage '55%' indicates that over this period of 3 seconds, our command 'wc' used an average of 55 percent of the available CPU cycles of the machine. This is a very high percentage and indicates that the system is lightly loaded.

The *unalias* and *unset* commands remove aliases and variable definitions from the shell.

The *wait* command can be used after starting processes with '&' to see quickly if they have finished. If the shell responds immediately with another prompt, they have. Otherwise you can wait for the shell to prompt, at which point they will have finished, or interrupt the shell by sending a RUBOUT or DELETE character. If the shell is interrupted, it prints the names and numbers of the processes it knows to be unfinished. Thus:

```
% nroff paper | lpr &
2450
2451
% wait
 2451  lpr
 2450  nroff
wait: Interrupted.
%
```

You can check again later by doing another *wait,* or see which commands are still running by doing a *ps.* As 'time' will show you, *ps* is fairly expensive. It is thus counterproductive to run many *ps* commands to see how a background process is doing.†

If you run a background process and decide you want to stop it you must use the *kill* program. You must use the process id number(s) (PIDs) of the process(es) you wish to kill. (If you don't know, do a *ps*). Thus to stop the *nroff* in the above pipeline you would do

```
% kill 2450
% wait
2450: nroff: Terminated.
%
```

Here the shell printed a diagnostic that we terminated 'nroff' only after we did a *wait.* If we want the shell to discover the termination of all processes it has created we must, in general, use *wait.*

7.10.7 What Else?

This concludes the basic discussion of the shell for terminal users. The programming features of the shell are described in the next section. One especially useful feature discussed later is the *foreach* built-in command, which can be used to run the same command sequence with a number of different arguments.

0. †If you do you are usurping with these *ps* commands the processor time the job needs to finish, thereby delaying its completion!

If you intend to use UNIX a lot, you should look through the rest of this document and the shell manual pages to become familiar with the other facilities available to you.

7.11 Shell Control Structures and Command Scripts

7.11.1 Introduction

Rather than inputting commands one at a time to the shell, you can place commands in files and cause these commands to be read and executed. Using these command files, you can put long command sequences in motion without having to wait for each one to complete serially; arithmetic and loop control constructs are also available. An added plus is that you can use all the expansion and substitution facilities of your editor to create these files quickly. These command files are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

7.11.2 Make

Note what shell scripts are *not* useful for. The program *make* is very useful for maintaining a group of related files or performing sets of operations on related files. For instance, a large program consisting of one or more files can have its dependencies described in a *makefile,* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily and most appropriately placed in this *makefile.* Shell scripts are less suitable for this kind of maintenance.

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

7.11.3 Invocation

A *csh* command script may be interpreted by saying

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

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

If you make the file 'script' executable by doing

```
chmod 755 script
```

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

```
script
```

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

A complication arises, however, when you run shell scripts under the C shell in UNIX SYSTEM III or Plexus Sys5. The C shell was written to be used with UNIX Version 7, the release *before* SYSTEM III, upon which Plexus Sys5 is based. Many standard Sys5 commands are actually shell scripts, and some begin with the comment character, so they look like C shell scripts to the C shell. But Sys5 contains commands not found in Version 7, so some of these shell scripts call commands the C shell has never heard of, and can't execute. Such commands fail when run under the C shell. Therefore, if you run the C shell with Sys5, you should set your environment variable SHELL to **/bin/sh**, so any shell scripts are automatically run by the Sys5 shell. If you really want the C shell to execute your shell script, you can explicitly execute it with 'csh' or put the call to 'csh' within the script.

7.11.4 Variable Substitution

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

```
echo $argv
```

when placed in a command script causes the current value of the variable *argv* to be echoed to the output of the shell script. *Argv* may not be unset at this point.

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

```
$?name
```

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

The notation

```
$#name
```

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

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

You can also 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'.

Other notations useful in shell scripts are

$*n*

(where *n* is an integer) as a shorthand for

$argv[*n***]**

(the *n th* parameter) and

$*

which is a shorthand for

$argv

The form

$$

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

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

Note also 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.

7.11.5 Expressions

Expressions in the shell must be able to be evaluated based on the values of variables; otherwise, the shell wouldn't be able to do anything interesting. All the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the Boolean 'and' and 'or' operations.

The shell also allows file enquiries of the form

–? filename

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

–e filename

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

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

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

7.11.6 Sample Shell Script

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

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

        if ($i:r.c != $i) continue          # not a .c file so do nothing

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

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

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

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

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

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

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

The placement of the keywords here is somewhat flexible.†

```
if ( expression )          # Produces diagnostic but works!
then
          command
          ...
endif
```

but this produces a diagnostic to the effect that the command 'then' cannot be found. The following format is not currently acceptable to the shell:

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

The 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 '\' to **immediately** precede the end-of-line.

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

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

else
          commands
endif
```

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

0. †Leaving out the 'then' altogether works, for example. The 'then' may also be put on a separate line, as follows:

```
    % echo $i $i:r
    foo.bar foo
    %
```

shows how the ':r' modifier strips off the trailing '.bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual entry. You can also use the *command substitution* mechanism described in section 4.3 to perform modifications on strings to reenter the shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive than the ':' modification mechanism.*

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

does not do what one would expect.

Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using ''' or '\' to place it in an argument word.

7.11.7 Other Control Structures

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

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

and

0. *Note also that the current implementation of the shell limits the number of ':' modifiers on a '$' substitution to 1. Thus

```
switch ( word )

case str1:
        commands
        breaksw

 ...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

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

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

```
loop:
        commands
        goto loop
```

7.11.8 Supplying Input to Commands

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

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

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

The notation '<< ´EOF´' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly '´EOF´'. The fact that the 'EOF' is enclosed in '´' characters, i.e. quoted, causes the shell not to perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' is quoted, then these substitutions are not performed. In this case, since we used the form '1,$' in our editor script, we needed to insure that this '$' was not variable-substituted. We could also have insured this by preceding the '$' here with a '\', i.e.:

```
1,\$s/↑[ ]*//
```

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

7.11.9 Catching Interrupts

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

```
onintr label
```

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

```
exit(1)
```

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

7.11.10 What Else?

Other features of the shell are useful to writers of shell procedures. The *verbose* and *echo* options and the related *–v* and *–x* command line options can help trace the actions of the shell. The *–n* option causes the shell only to read commands and not to execute them.

Note that *csh* does not execute shell scripts that do not begin with the character '#'--that is, shell scripts that do not begin with a comment.

Another quotation mechanism using "" allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as '' does. See the manual entry for more information.

7.12 Miscellaneous, Less Generally Useful, Shell Mechanisms

7.12.1 Loops at the Terminal; Variables as Vectors

You may occasionally want to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, three shells were once in use on the Cory UNIX system at Cory Hall at UC Berkeley: '/bin/sh', '/bin/nsh', and '/bin/csh'. To count the number of persons using each shell, one could issue the commands

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

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

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

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

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

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

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

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

7.12.2 Braces { ... } in Argument Expansion

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

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

expands to

```
Astr1B Astr2B ... AstrnB
```

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

A typical use of this would be

```
mkdir /{hdrs,retrofit,csh}
```

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

```
chown bin /usr/{bin/{ex,edit},lib/{ex1.1strings,how_ex}}
```

This command changes the ownership of all the following files: /usr/bin/ex, /usr/bin/edit, /usr/lib/ex1.1strings, and /usr/lib/how_ex.

7.12.3 Command Substitution

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

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

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

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

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

7.12.4 Other Details Not Covered Here

Sometimes you may need to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in the manual section.

The shell has a number of command line option flags that are mostly of use in writing UNIX programs and debugging shell scripts. See the manual entry for a list of these options.

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

7.13 Appendix – Special Characters

The following table lists the special characters of *csh* and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *csh* manual entry for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
|	1.5	separates commands in a pipeline
( )	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[ ]	1.6	expansion sequence matching any single character from a set
	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
´	1.7	prevents meta-meaning of a group of characters
"	4.3	like ´, but allows variable and command expansion

Input/output metacharacters

<	1.3	indicates redirected input
>	1.5	indicates redirected output

Expansion/substitution metacharacters

$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	3.6	begins a shell comment
–	1.2	prefixes option (flag) arguments to commands

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of this document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the *Plexus Sys5 UNIX Programmer's Reference Manual* in section 1. You can get an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual. Your current directory has the name '.' as well as the name printed by the command *pwd.* The current directory '.' is usually the first component of the search path contained in the variable *path;* thus commands that are in '.' are found first (2.2). The character '.' is also used in separating components of filenames (1.6). The character '.' at the beginning of a component of a pathname is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs (1.6). Each directory has a file '..' in it, which is a reference to its *parent* directory. After changing into the directory with *chdir,* i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by *pwd* (2.6). An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias,* which establishes aliases and can print their current values. The command *unalias* is used to remove aliases (2.6). Commands in UNIX receive a list of argument words. Thus the command

```
echo a b c
```

consists of a command name 'echo' and three argument words 'a', 'b' and 'c' (1.1). The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4). Commands started without waiting for them to complete are called *background* commands (1.5). A directory containing binaries of programs and shell scripts to be executed is typically called a 'bin' directory. The standard system 'bin' directories are '/bin', which contains the most heavily used commands, and '/usr/bin', which contains most other user programs. Other binaries are contained in directories such as '/usr/plx' where new and Plexus-specific programs are placed. You can place binaries in any directory. If you wish to execute them often, the directory's name should be a component of the variable *path. Break* is a built-in command used to exit from loops within the control structure of the

shell (3.6). A command executed directly by the shell is called a *built-in* command. Most commands in UNIX are not built-into the shell, but rather exist as files in 'bin' directories. These commands are accessible because the directories in which they reside are named in the *path* variable. A *case* command is used as a label in a *switch* statement in the shell's control structure, similar to that of the language C. Details are given in the shell's documentation 'csh (NEW)' (3.7). The *cat* program catenates a list of specified files on the standard output. It is usually invoked to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3). The *cd* command is used to change the working directory. With no arguments, *cd* changes your working directory to be your *home* directory (2.3) (2.6). The *chdir* command is a synonym for *cd. Cd* is usually typed because it is shorter. *Cmp* is a program that compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff,* described in 'diff (1)' is used. A function performed by the system, either by the shell (a built-in command) or by a program residing in a file in a directory within the UNIX system is called a *command* (1.1).
The replacement of a command enclosed in '`' characters by the text output by that command is called *command substitution* (3.6, 4.3). A part of a *pathname* between '/' characters is called a *component* of that pathname. A *variable* that has multiple strings as value is said to have several *components,* and each string is a *component* of the variable. A built-in command that causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6). When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This 'core dump' can be examined with the system debuggers 'adb (1)' and 'sdb (1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form:

commandname: Illegal instruction — Core dumped

(where 'Illegal instruction' is only one of several possible messages) you should report this to the author of the program and save the 'core' file. The *cp* (copy) program copies the contents of one file into another file. It is one of the most commonly used UNIX commands (2.6). The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters, which take effect globally (2.1). The *date* command prints the current date and time (1.3). *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables that may be used to aid in shell debugging (4.4). The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7). The DELETE or RUBOUT key on the terminal is used to generate an INTERRUPT signal, which stops the execution of most programs on UNIX (2.6). A command run without waiting for it to complete is said to be detached (2.5). An error message produced by a program is often referred to as a *diagnostic.* Most error messages are not written to the standard output, since that is often directed away from the terminal (1.3, 1.5). Error messages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus diagnostics will usually appear on the terminal (2.5). A structure that contains files. At any time you are 'in' one particular directory whose names can be printed by the command 'pwd'. The *chdir* command will position you 'in' another directory. The directory in which you are when you first login is your *home* directory (1.1, 1.6). The *echo* command prints its arguments (1.6, 2.6, 3.6, 3.10). The *else* command is part of the 'if-then-else-endif' control command construct (3.6). An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file that it has been given as input. Commands receiving input from a *pipe* receive an end-of-file when the command sending them input completes. Most commands terminate when they receive an end-of-file. The shell has an option to ignore end-of-file from a terminal input; this option may help you keep from logging out accidentally (1.1, 1.8, 3.8). A character \ used to prevent the special meaning of a metacharacter is called an *escape* character, because the special meaning is 'escaped'. Thus

```
echo \*
```

will echo the character '*' while just

```
echo *
```

will echo the names of the file in the current directory. In this example, \ *escapes* '*' (1.7). There is also a non-printing character called *escape,* usually labeled ESC or ALTMODE on terminal keyboards. Some UNIX systems use this character to indicate that output is to be suspended. Other systems use control-s. This file contains information about the accounts currently on the system. If consists of a line for each account with fields separated by ':' characters (2.3). You can look at this file by saying

```
cat /etc/passwd
```

The command *grep* is often used to search for information in this file. See 'passwd (5)' and 'grep (1)' for more details. The *exit* command is used to force termination of a shell script, and is built-into the shell (3.9). A command that discovers a problem may reflect this back to the command (such as a shell) that invoked (executed) it. It does this by returning a non-zero number as its *exit status,* a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script to give a non-zero exit status (3.5). The replacement of strings that

contain metacharacters by other strings is referred to as the process of *expansion.* Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. Expansions are also referred to as *substitutions* (1.6, 3.4, 4.2). Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell expressions are those of the language C (3.5). Filenames often consist of a *root* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same root name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '–me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6). Each file in UNIX has a name consisting of up to 14 characters and not including the character '/'. This name is used in *pathname* building. Most file names do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the root portion of the filename from an extension (1.6).
Filename expansion uses the metacharacters '*', '?' and '[' and ']' to provide a convenient mechanism for naming files. Using filename expansion it is easy to name all the files in the current directory, or all files that have a common root name. Other filename expansion mechanisms use the metacharacter ' ' and allow files in other users directories to be named easily (1.6, 4.2). Many UNIX commands accept arguments that are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consist of one or more letters preceded by the character '–' (1.2). Thus the *ls* list file commands has an option '–s' to list the sizes of files. This is specified

```
ls –s
```

The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1). The *getty* program is part of the system that determines the speed at which your terminal is to run when you first log in. It types the initial system banner and 'login:'. When no one is logged in on a terminal a *ps* command shows a command of the form '- 7' where '7' here is often some other single letter or digit. This '7' is an option to the *getty* command, indicating the type of port it is running on. If you see a *getty* command running on a terminal in the output of *ps* you know that no one is logged in on that terminal (2.3). The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7). The *grep* command searches through a list of argument files for a specified string. Thus

grep bill /etc/passwd

will print each line in the file '/etc/passwd' that contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed (1)' and 'ex (1)' (2.3). *Grep* stands for 'globally find regular expression and print' or, some say, 'get regular expression pattern'. When you hangup a phone line, a HANGUP signal is sent to all running processes on your terminal, causing them to terminate execution prematurely. If you wish to start commands to run after you log off a dialup you must use the command *nohup* (2.6). The *head* command prints the first few lines of one or more files. If you have files containing text you are wondering about, try running *head* with these files as arguments. This usually shows enough of what is in these files to let you decide which you are interested in (1.5, 2.3). The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable, which controls how large this list is (1.7, 2.6). Each user has a home directory, which is given in your entry in the password file, */etc/passwd.* This is the directory you are placed in when you first log in. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *HOME.* You can also access the home directories of other users in forming filenames using a file expansion notation and the character ' ' (1.6). A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6). Normally, your shell exits, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. But you can *set* the *ignoreeof* variable in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2, 2.6). Many commands on UNIX take information from the terminal or from files that they then act on. This information is called *input.* Commands normally read for input from their *standard input*, which is, by default, the terminal. This standard input can be redirected from a file using a shell metanotation with the character '<'. Many commands also read from a file specified as argument. Commands placed in pipelines read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a file name to use as standard input. Special mechanisms exist for supplying input to commands in shell scripts (1.2, 1.6, 3.8). An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key. It causes most programs to stop execution. Certain programs such as the shell and the editors handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you hit interrupt because many commands die when they receive an interrupt (1.8, 2.6, 3.9). A program that terminates processes run without waiting for them to complete. (2.6) The file *.login* in your *home* directory is read by the shell each time you log in to UNIX and the commands there are executed. A number of commands are usefully placed here, especially *tset* commands and *set* commands to the shell itself (2.1). The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an end-of-file, but if you have set *ignoreeof* in your *.login* file, then this will not work and you must use *logout* to log off the UNIX system (2.2). When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'. The command *lpr* is the line printer daemon. The standard input of *lpr* is spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. *Lpr* is usually the last component of a *pipeline* (2.3). The *ls* list files command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2). The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.2). The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2). The file containing commands for *make* is called 'makefile' (3.2). The 'manual' often referred to is the *Plexus Sys5 UNIX Programmer's Reference Manual.* It contains a description of each UNIX program. An online version of the manual is accessible through the *man* command. Its documentation can be obtained online via

```
        man man
```

Many characters that are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters.* If these characters must be used without their special meaning in arguments to commands, then they must be *quoted.* An example of a metacharacter is the character '>', which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted metacharacters form separate words (1.4). The appendix to this manual lists the metacharacters in groups by their function. The *mkdir* command is used to create a new directory (2.6). Substitutions with the history mechanism, keyed by the character '!' or of variables using the metacharacter '$' are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the modifier itself. The *command* *substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6). The shell has a variable *noclobber,* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5). A shell command used to allow background commands to run to completion even if you log off a dialup before they complete. (2.5) The standard text formatter on UNIX is the program *nroff.* Using *nroff* and one of the available *macro* packages for it, documents may be automatically formatted and prepared for phototypesetting using the typesetter program *troff* (3.2). The *onintr* command is built-into the shell and is used to control the action of a shell command script when an interrupt signal is received (3.9). Many commands in UNIX produce *output.* This output is usually placed on what is known as the *standard output*, which is normally connected to the user's terminal. The shell metacharacter '>' redirects the standard output of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|', the standard output of one command may become the standard input of another command (1.5). Certain commands such as the line printer daemon *lpr* do not place their results on the standard output but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another user's terminal rather than its standard output (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the standard output has been sent to a file or another command, but you may direct error diagnostics along with standard output using a special metanotation (2.5). The shell has a variable *path,* which gives the names of the directories in which it searches for the commands that it is given. It always checks first to see if the command it is given is built-into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not built-in, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

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

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

```
chmod 755 script
```

was executed to turn these execute bits on (3.3). A list of names, separated by '/' characters forms a *pathname.* Each *component,* between successive '/' characters, names a directory in which the next component file resides.

Pathnames that begin with the character '/' are interpreted relative to the *root* directory in the file system. Other pathnames are interpreted relative to the current directory as reported by *pwd.* The last component of a pathname may name a directory, but usually names a file. A group of commands connected together, the standard output of each connected to the standard input of the next, is called a *pipeline.* The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|' (1.5, 2.3). The *pr* command prepares listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3). The *printenv* command is used on version 7 UNIX systems to print the current setting of variables in the environment (2.6). A instance of a running program is called a process (2.6). The numbers used by *kill* and printed by *wait* are unique numbers generated for these processes by UNIX. They are useful in *kill* commands, which can be used to stop background processes. (2.6) Usually synonymous with *command;* a binary file or shell command script that performs a useful function is often called a program.
Also referred to as the *manual.* See the glossary entry for 'manual'. Many programs print a prompt on the terminal when they expect input. Thus the editor 'ex' prints a ':' when it expects input. The shell prompts for input with '% ' and occasionally with '? ' when reading commands from the terminal (1.1). The shell has a variable *prompt,* which may be set to a different value to change the shell's main prompt. This is mostly used when debugging the shell (2.6). The *ps* command shows the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.3, 2.6). Login shells, such as the *csh* you get when you login are shown as '–'. The *pwd* command prints the full pathname of the current (working) directory. The *quit* signal, generated by a control-\ is used to terminate programs that are behaving unreasonably. It normally produces a core image file (1.8). The process by which metacharacters are prevented their special meaning, usually by using the character '' in pairs, or by using the character '\' (1.4). The routing of input or output from or to a file is known as *redirection* of input or output (1.3). The *repeat* command iterates another command a specified number of times (2.6). The RUBOUT or DELETE key generates an interrupt signal that is used to stop programs or to return and prompt for more input (2.6). Sequences of shell commands placed in a file are called shell command scripts. You may perform simple tasks using these scripts without writing a program in a language such as C, by using the shell to selectively run other programs (3.2, 3.3, 3.10). The built-in *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using

the set command the behavior of the shell can be affected (2.1). Variables in the environment 'environ (5)' can be changed by using the *setenv* built-in command (2.6). The *printenv* command can be used to print the value of the variables in the environment. A shell is a command language interpreter. You may write and run your own shell, as shells are no different from any other programs as far as the system is concerned. This manual deals with the details of one particular shell, called *csh.* See *script* (3.2, 3.3, 3.10). The *sort* program sorts a sequence of lines in ways that can be controlled by argument flags (1.5). The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.6).
See *metacharacters* and the appendix to this manual. We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8). A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The shell variable *status* is set to the status returned by the last command. It is most useful in shell command scripts (3.5, 3.6). The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter '!' and variable substitution indicated by '$'. We also refer to substitutions as *expansions* (3.4). The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7). When a command that is being executed finishes, we say it undergoes *termination* or *terminates.* Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified command whose numbers are given (2.6). The *then* command is part of the shell's 'if-then-else-endif' control construct used in command scripts (3.6). The *time* command can be used to measure the amount of CPU and real time consumed by a specified command (2.1, 2.6). The *troff* program is used to typeset documents. See also *nroff* (3.2). The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1). The *unalias* command removes aliases (2.6). UNIX is an operating system on which *csh* runs. UNIX provides facilities that allow *csh* to invoke other programs such as editors and text formatters. The *unset* command removes the definitions of shell variables (2.2, 2.6).
See *variables* and *expansion* (2.2, 3.4). Variables in *csh* hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See *path, noclobber,* and *ignoreeof* for examples. Variables such as *argv* are also used in writing shell programs (shell

command scripts) (2.2). The *verbose* shell variable can be set to cause commands to be echoed after they are history-expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shells *–v* command line option (3.10). The built-in command *wait* causes the shell to pause, and not prompt, until all commands run in the background have terminated (2.6). The *while* built-in control construct is used in shell command scripts (3.7). A sequence of characters that forms an argument to a command is called a *word.* Many characters that are neither letters, digits, '–', '.' or '/' form words all by themselves even if they are not surrounded by blanks. Any sequence of character may be made into a word by surrounding it with '´' characters except for the characters '´' and '!', which require special treatment (1.1, 1.6). This process of placing special characters in words without their special meaning is called *quoting.*
At an given time you are in one particular directory, called your working directory. This directory's name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change working directories using *chdir.* The *write* command is used to communicate with other users who are logged in to UNIX (2.3).

The following list defines terms and acronyms used in this volume which may not be familiar to the user.

argument—Words following the command on a command line that provide information necessary to execute a program. Command arguments are very often file names.

ASCII—American Standard Code for Information Interchange.

background—A mode of program execution when the shell does not wait for the command to terminate before prompting for another command.

C language—A general purpose low level programming language used to write programs (such as numerical, text-processing, and data base) and operating systems (such as the UNIX operating system).

command—The first word of a command line. It is the name of an executable file that is a compiled program.

command line—A sequence of nonblank arguments separated by blanks or tabs typed in by a user. The first argument usually specifies the name of a command.

command list—A sequence of one or more simple commands separated or terminated by a new line or a semicolon.

command procedure—A command procedure is an executable file that is not a compiled program. It is a call to the **shell** to read and execute commands contained in a file. A sequence of commands may thus be perserved for repeated use by saving it in a file which can also be called a shell procedure, a command file, or a runcom according to local preference.

command substitution—When the **shell** reads a command line, any command or commands enclosed between grave accents ('...') are executed first and the output from these commands replace the whole expression ('...').

current working directory—The current point of reference for accessing data within the file system.

directory—A type of file that is used to group and organize files and other directories.

EOF—The End-Of-File character is the same as an ASCII EOT character.

See EOT.

EOT—The End-Of-Text character is generated by holding down the "CONTROL" key and pressing the lowercase "d" key once. The EOT is used to terminate the **shell** which usually logs a user off the system.

erase character—The character which is used to delete the previous character on the current line. To turn off the special meaning of the erase character, it must be preceded with a "\". By default, the erase character is #. See **stty**(1) to change the default character.

file—An organized collection of information containing data, programs, or both which allows users to store, retrieve, and modify information. A simple file name is a sequence of characters other than a slash (/).

filter—A command that reads its standard input, transforms it in some way, and prints the result as output.

foreground—A mode of program execution when the shell waits for the command to terminate before prompting for another command.

full pathname—The pathname of a specific file starting from the **root** directory.

group identification number (gid)—A unique number assigned to one or more logins that is used to identify groups of related users.

here documents—A command procedure that has the form **command** << **eofstring** which causes the **shell** to read subsequent lines as standard input to the command until a line is read consisting of only the **eofstring**. Any arbitrary string can be used for the **eofstring**.

HOME—Another name for the login directory.

in-line input documents—See here documents.

keyword parameters—An argument to a command procedure of the form **name=value command arg1 arg2 . . .** here **name** is called the keyword parameter. This allows **shell** variables to be assigned values when a **shell** procedure is called. The value of **name** in the invoking **shell** is not affected, but the value is assigned to **name** before execution of the procedure. The arguments (arg1 arg2 . . .) are available as positional parameters ($1 $2 . . .).

kill character—The character which is used to delete all the characters typed before it on the current line. To turn off the special meaning of the kill character, it must be preceded with a "\". By default, the kill character is @. The default character can be changed via **stty**(1).

login—A means by which a user can gain access to the UNIX operating system.

login name—A unique string of letters and numbers used to identify a login.

log off—A procedure to disconnect the user from the UNIX operating system.

memorandum macros—The standard general purpose package of text formatting macros used in conjuction with nroff and troff to produce many common types of documents.

metacharacters—Characters that have a special meaning to the **shell**, such as <, >, *, ?, | &, $, ;, (,), \, ", ‘, ´, [,], etc.

mode—An absolute mode is an octal number used in conjunction with **chmod**(1) to change permissions of files.

nroff—A text formatting program for driving typewriter-like terminals and printers to produce a screen copy or a hardcopy.

parent directory—The directory immediately above another directory. A ".." is the shorthand name for the parent directory. To make the parent directory of your current working directory your new current directory enter the "**cd ..**" command.

partial pathname—The pathname between the current working directory and a specific file.

password—A string of up to 13 characters chosen from a 64 character alphabet (., \, 0-9, A-Z, a-z).

pathname—A sequence of directory names separated by the / character and ending with the name of a file. The pathname defines the connection path between some directory and a file.

pipe—A simple way to connect the output of one program to the input of another program, so that each program will run as a sequence of processes.

pipeline—A series of filters separated by the character |. The output of each filter becomes the input of the next filter in the line. The last filter in the line will write to its standard output.

positional parameters—Arguments supplied with a command procedure that are placed into variable names **$1, $2, . . .** when the command procedure is invoked by the **shell**. The name of the file being executed is positional parameter **$0**.

primary prompt—A notification (by default "**$** ") to the user that the UNIX operating system **shell** is ready to accept another request.

process—A program that is in some state of execution. The execution of a computer environment including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items.

program—Software that can be executed by a user.

secondary prompt—A notification (by default "> ") to the user that the command typed in response to the primary prompt is incomplete.

shell—A UNIX system user program written in C language that handles the communication between the system and users. The shell accepts commands and causes the appropriate program to be executed.

shell procedure—See command procedure.

standard input—The standard input of a command is sent to an open file which is normally connected to the keyboard. An argument to the **shell** of the form "< file" opens the specified file as the standard input thus redirecting input to come from the file named instead of the keyboard.

standard output—Output produced by most commands is sent to an open file which is normally connected to the printer or screen. This output may be redirected by an argument to the **shell** of the form "> file" which opens the specified file as the standard output.

text editor—An interactive program (**ed**) for creating and modifying text, using commands provided by a user at a terminal.

troff—A text formatting program for driving a phototypesetter to produce high quality printed text.

user-defined variables—A user variable can be defined using an assignment statement of the form **name=value** where **name** must begin with a letter or underscore and may then consist of any sequence of letters, digits, or underscores up to 512 characters. The **name** is the variable. Positional parameters cannot be in the name.

user identification number (uid)—A unique number assigned to each login that is used to identify users and the owner of information stored on the system.

variables—A variable is a name representing a string value. Variables which are normally set only on a command line are called parameters (positional parameters and keyword parameters). Other variables are simply names to which the user (user-defined variables) or the **shell** itself may assign string values.