sun®
microsystems

Doing More *with* UNIX: Beginner's Guide

Part No: 800-12[illegible]-05
Revision A of 17 February 1986

Credits and Trademarks

Contents

Tables

Figures

Preface

This manual describes some of the more sophisticated features UNIX† provides, and how to use them to simplify complicated tasks.

Chapter 1 is a brief introduction.

Chapter 2 provides details about files, their attributes, filename substitution, and searching through text files.

Chapter 3 describes how to use commands as building blocks for complicated tasks.

Chapter 4 provides an overview of the C-Shell and its timesaving features.

Chapter 5 describes processes and their behind-the-scenes role in providing balanced service to concurrent tasks.

Chapter 6 introduces tools for sophisticated file management.

Chapter 7 describes the printer queue, how to select a printer, printing preformatted files, and printing graphics from the workstation screen.

In addition to a glossary, command summary, and quick reference, there are appendices that describe details about the C-Shell, such as C-Shell special commands (called "builtin" commands), predefined variables, special characters and scripts.

Prerequisite Documents	*Getting Started With UNIX: Beginner's Guide* *Setting Up Your UNIX Environment: Beginner's Guide* *Self Help With Problems: Beginner's Guide*
Companion Documents	*Using the Network: Beginner's Guide* *Commands Reference Manual for the Sun Workstation*

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

1

Introduction

1

Introduction

UNIX† provides you with features that are powerful, flexible, and adaptable. This means that there is quite a lot that the system can do for you, and there is quite a lot to learn. The power and richness of the commands make for limitless possibilities. In fact, one of the main advantages of the UNIX system design is its open-ended nature.

Everyone goes through several stages when learning to use UNIX effectively, including:

a) learning the basics

You are here. ⇒

b) learning enough to get curious

c) experimenting with the various features and commands

d) educated experimentation and writing simple shell scripts

e) digging deeper into the system and its internal workings.

This manual is intended to help satisfy your curiosity with an overview of features that give you major productivity gains.

Previous manuals in this series, such as *Getting Started with UNIX*, *Setting Up the UNIX Environment*, and *Games, Demos and Other Pursuits*, gave you a basic familiarity with UNIX, but may not have answered questions about *why* the system works the way it does, or *how* to get more out of it. Hopefully, this one does.

Companion manuals, such as *Using the Network* will tell you about more specialized topics.

Why and How

From its origins as a simple research project, UNIX has evolved into a powerful, flexible and popular computer operating system, and a major influence in the industry. It was designed to accommodate this evolution by providing a simple model for storing and transferring information, called a *file*, a collection of simple commands to operate on files, and a straightforward method for combining commands to perform more complicated tasks. Because UNIX grew out of a computer science research environment, the terminology and command names are oriented toward professionals in that field, as are many of the tools.

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

Commands are terse to save keystrokes. They are usually suggestive of the simple function they perform. Unless you are already familiar with those sorts of functions, the names may seem cryptic. The more you learn, the more sensible things will begin to seem. So, rather than being put off by it, get familiar with the jargon! You'll learn a lot more about computers than just how to use one.

Try it Yourself!

When learning more about UNIX, there is no substitute for experimenting on your own. To really grasp what a command does, you simply have to try it. So, as you go through this, and the remaining beginner's guides in this series, try out the examples. Then try out variations of your own design.

Play it Safe!

Whenever you experiment with UNIX it is important to set up a safe place in which to do so. Never experiment with an unfamiliar command on valuable data. Instead, make a copy and place it in a directory where the data is known to be dispensable. Always run your tests in this directory to avoid the risk of corrupting previous work. Once you have tested the command and have seen what it does, only then should you apply it to files that you care about.

Make a directory, `test`, in your home directory, as follows:

```
mars% cd
mars% mkdir test
```

Consider everything in this directory to be expendable, and never place anything there that you intend to keep.

Hang in There!

Because UNIX was developed to support programming research, its standard features are oriented toward the programming professional. This is one reason why the system is so powerful, and also why some features seem a bit abstract at first. In most cases, their power and flexibility make this an easy thing to get used to.

UNIX is designed to be general in scope. It can support a wide variety of applications, and work well within a broad range of situations. The information in this manual should help you to take this general and flexible, but somewhat abstract system, and use it to meet your specific needs and working style.

2

More About Files

2

More About Files

2.1. Filename Substitution

As you learned in *Getting Started With UNIX*, filename wild cards can save you time and keystrokes. The system replaces, or *substitutes* characters from filenames for the wild card symbols.

I.n addition to the wild cards, `*`, and `?`, UNIX provides more sophisticated ways of specifying a set of files on the command line.

Single-Character Matching with `[` and `]`

You can use *brackets* instead of a `?`, to match a single character. Within the brackets you can specify a list of characters to match against. For instance,

```
[ab]*
```

matches all filenames that begin with a lower-case `a` or `b`. You can also specify a *range* of characters to match against. Thus,

```
[A-Z]*
```

matches all filenames that begin with an upper-case alphabetical character.

Listing Hidden Files with `ls -a`

Filenames that begin with a dot (`.`) are a special case. They aren't matched unless you specify a dot in the first character. However, the name `.` stands for the current directory, and `..` stands for the parent directory. So, although the command

```
ls .*
```

does list hidden files, it *also* lists all the other files in the directory (matching `./*`), and the parent directory (matching `../*`).

To list hidden files along with the others, use the command:

```
ls -a
```

String Matching with `{` and `}`

You can use *braces* instead of `*`, to match specific character strings of any length. Within the braces, strings are separated by commas. For instance,

```
{uranus,sygnus,x}*
```

matches any filenames beginning with `uranus`, `sygnus` or `x`.

Within braces, `*`, and `?`, are legal. You can nest braces within strings for interesting results. For instance, `{{ura,syg}nus,x}*` is another way to match filenames beginning with `uranus`, `sygnus` or `x`.

2.2. Properties of Files

As your skill with the system grows, you will encounter situations in which a prior understanding of files and their properties, especially file *ownership* and *permissions*, will be of immense help.

You can think of a file as a named location from which infomation can be obtained or to which data can be sent. UNIX uses the notion of a file as a general model for all sources (input) or destinations (output) of data operated on by commands. The system treats terminals, printers, tape drives, and other such devices for putting information into, or getting information out of the system, as if they too were *files*.

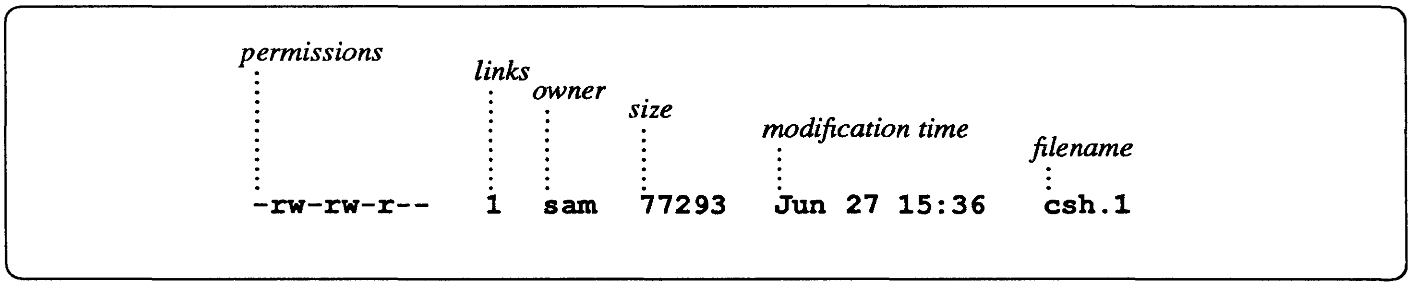
Commands and programs don't need to know whether the data they use comes from (or goes to) a terminal, disk file, printer (or even another program). Just like any other file, each device has a pathname. The `tty` command tells you the pathname of your terminal or window.

```
mars% tty
/dev/ttyp1
mars%
```

In addition to having a *name*, and *contents*, a file under UNIX has other important properties that you can examine with options to `ls`. (Refer to `ls` in the *Commands Reference Manual* for a complete list of these options.) The `-l` options shows a more detailed (long) list of the files:

```
mars% ls -l
total 112
-rw-rw-r--  1 sam        77293 Jun 27 15:36 csh.1
-rw-rw-r--  1 wild       27492 Jul  9 21:14 csh.blt
-rw-rw-r--  1 ames        6550 Jul  9 21:02 csh.new
-rw-rw-r--  1 root       14492 Jul 12 17:07 csh.spc
-rw-r--r--  1 sam         2884 Jul 17 18:24 files
-r-xr-xr-x  1 sam         1381 Jul 12 15:50 script
mars%
```

The top line tells you how many blocks (units of space on the disk), are occupied by files in the directory. The remaining lines are composed of columns that describe specific properties of each file:



```
-rw-rw-r--   1  sam  77293  Jun 27 15:36   csh.1
```

Figure 2-1 *Information Displayed By* `ls -l`

The leftmost column shows the *permissions* for each file. Permissions are explained in detail below. The second column shows the number of *links*, to it. Links are also described later on.

The third column shows each file's *owner*. Normally, the owner of a file is the person who created it, although the operator of your system can change this. Not shown here is the file's *group* ownership.

The fourth column shows the file's *size* in bytes. The size of the file often changes when you edit it. The next three columns show the date and time when the file was last modified (*modification time*). This also changes whenever you edit the file.

The rightmost column shows the *filename*.

2.3. Permissions

Every file has a set of access modes or *permissions* that determine which users have access to read, write, or *execute* its contents.

Like devices, programs are treated as files. When you enter a command, UNIX looks up a file by that name among the directories listed in the `PATH` environment variable, and performs the instructions contained in that file.

The *permissions* column consists of ten characters as shown in Figure 2-1, above. The leftmost character shows the type of file (regular, directory or device). The next triplet of characters displays access modes for the owner. The second triplet shows those for the group, and the last, those for the public.

File Type

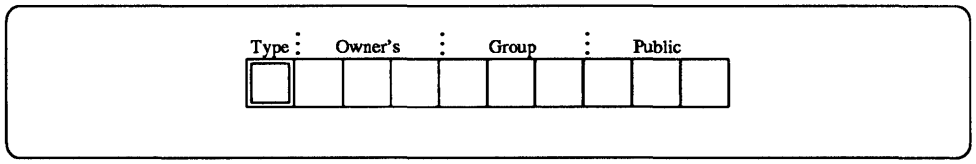


Figure 2-2 *The File Type Field*

A `d` in the leftmost character indicates that the file is a directory. A – indicates a standard file. A `b`, or `c` indicates that the file is a *device*. An `s`, indicates that the file is a *socket* for communication between two running programs. An `l` indicates that the filename is a *symbolic link* that refers to the name of another file.

Owner's Permissions

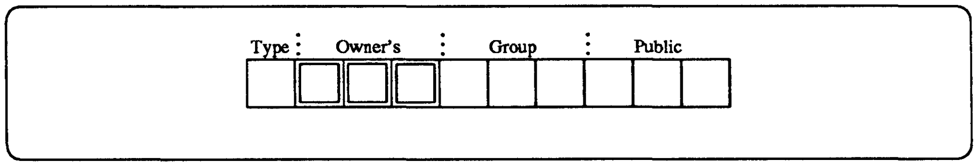


Figure 2-3 *Owner's Permissions Field*

In the listing of Figure 2-1, `sam` is the owner of the file `csh.1`. An `r` as the first character in this triplet indicates that the owner has permission to read the file. A – indicates that the permission does not apply. A `w` as the second character indicates that the owner can write on (modify, add to, or remove) the file. An `x` as the third character indicates that the owner can execute the file (use it as if it were a command[1]).

Of course, unless the file is either a program or list of shell commands, executing it doesn't make any sense.

As Figure 2-1 shows, `sam` can read and write on, but not execute the file `csh.1`.

Group Permissions

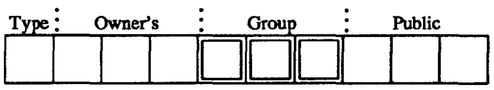


Figure 2-4 *Group Permissions Field*

To see which *group* the file belongs to, use the `-lg` option of `ls`.

```
mars% ls -lg
total 112
-rw-rw-r--  1 sam    wheel   77293 Jun 27 15:36 csh.1
-rw-rw-r--  1 wild   wheel   27492 Jul  9 21:14 csh.blt
-rw-rw-r--  1 ames   wheel    6550 Jul  9 21:02 csh.new
-rw-rw-r--  1 root   wheel   14492 Jul 12 17:07 csh.spc
-rw-r--r--  1 sam    wheel    2884 Jul 17 18:24 files
-r-xr-xr-x  1 sam    wheel    1381 Jul 12 15:50 script
mars%
```

In this case, all files belong to the group `wheel`. The files `csh.1` through `csh.spc` can be read and written on by any member of the group. The file `script` can be executed and read, but not written on.

Public Permissions

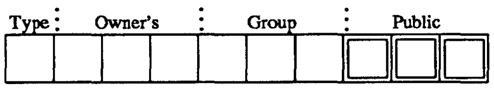


Figure 2-5 *Public Permissions Field*

All files in the above list can be read by anyone. The x in the rightmost character for `script` indicates that anyone can use it as a command.

Permissions of Directories

With directories, the access modes have a slightly different meaning. To check the permissions of the current directory, use the `-ld` option of `ls`.

```
mars% ls -ld
drwxrwxr-x  3 sam          512 Jul 16 23:10
mars%
```

An r indicates that the directory can be *read*. You must have read access to a directory before you can list its contents or cd into it.

An x indicates that the directory can be *searched* (that you can list its contents). A w indicates that files can be added or removed from the directory.

You can remove any file in a directory for which you have write permission, regardless of who owns that file. If you do not have write permission for the file itself, the system asks you for confirmation before removing it.

In the directory shown above, the owner (sam) can read, search, and add or delete files, as can the group. The public can read and search, but cannot add or delete files.

2.4. Changing Permissions with chmod

From time to time you may want to change the access modes of files that you own, either to restrict or to allow access to it. In most cases, restricting access to a file is sufficient to protect it from tampering or unwarranted reading. Even so, you should be aware that the operator of your system has unlimited access to any file. Because UNIX evolved in a relatively friendly research-and-development setting, the file system provides adequate, but not unbreakable, security between users.[2]

You can use an argument to chmod to specify the access mode for each class of user (owner, group, or public), or to indicate how the mode is to be changed. An argument is composed of one or more classes, an operation, and one or more permissions from the chart below:

Table 2-1 chmod *Command Syntax Diagram*

chmod [*class(es)*] *operation permission(s)* [, ...] *filename* ...

where *class(es)*, *operation* and *permission(s)* can be selected from:

class		*operation*		*permission*	
u	user (owner)	=	set permission	r	read
g	group	–	remove access	w	write
o	others (public)	+	give access	x	execute
a	all				

For example, the command

```
mars% chmod o-r,a+x,g=rw csh.1
mars%
```

[2] No computer system provides unbreakable security between authorized users. Also note that the system administrator can read any file on the system. If you want to protect your files from unauthorized reading, you can *encrypt* them. See *Encrypting Files* below, for details.

a) removes read permission for the public (others),

b) adds execute permission for all three classes, and

c) sets access to read and write for the group

for the file `csh.1`.

If you omit *class*, the new setting is applied to all three.

`chmod` can also use a digit from zero to seven to represent each triplet in the permissions column, as follows:

chmod [*o*[*g*]]*p*

where *o* is a digit representing the owner's permissions, *g* is a digit representing the group permissions, and *p* is a digit representing permissions for the public. The value of each digit is the sum of the permission values as in the following chart.

Table 2-2 *Chart of* `chmod` *Numeric Arguments*

value	*permission*	*explanation*
4	`r`	read
2	`w`	write
1	`x`	execute

To figure each digit, add up the values corresponding to each permission setting in the triplet. For read, write and execute permission, the value is 7. All values, and the permissions they correspond to, are shown below:

value	*permissions*	*explanation*
7	`rwx`	read, write, and execute
6	`rw-`	read and write
5	`r-x`	read and execute
4	`r--`	read only
3	`-wx`	write and execute
2	`-w-`	write only
1	`--x`	execute only
0	`---`	no access whatsoever

The command

```
mars% chmod 777 csh.1
mars%
```

gives read, write and execute access to `csh.1` to the owner, the group, and the public.

On the other hand, the command

```
mars% chmod 7 csh.1
mars%
```

gives the public read and write access, and denies all access to the owner and the group. So, although they aren't required, it's a good idea always to use all three digits.[3]

2.5. Setting Default Permissions with `umask`

When you create a new file or directory, the system automatically assigns permissions. The default setting for new files is

```
-rw-r--r--
```

or `644`. For new directories, the default is

```
drwxr-xr-x
```

or `755`.

You can change the default permission setting for the current session with the `umask` command:

`umask` [*o*[*g*]]*p*

o, *g* and *p* are digits corresponding to the owner's, group, and public permission masks, respectively.

You can change the permissions for all sessions by placing a `umask` command in your `.cshrc` file.

Like `chmod`, `umask` uses three digits to determine the permissions. Unlike, `chmod`, it computes the permissions according to the following table:

Table 2-3 `umask` *Values and Resulting Permissions for New Files*

Files		*Directories*	
value	*permissions*	*value*	*permissions*
0	rw-	0	rwx
1	rw-	1	rw-
2	r--	2	r-x
3	r--	3	r--
4	-w-	4	-wx
5	-w-	5	-w-
6	---	6	--x
7	---	7	---

`umask` does not activate *execute* permission for files.

So, the command

```
umask 2
```

or

[3] There is also a fourth digit, one that is used to allow certain programs to assume another user ID or group ID while running, or to to remain in memory even when stopped. Unless you are writing programs like that, you will have little occasion to use the fourth digit.

```
umask 002
```

yields permissions of `-rw-rw-r--` for files, and `drwxrwxr-x` for directories.

The command

```
umask 22
```

yields permissions of `-rw-r--r--` for files and `drwxr-xr-x` for directories.

2.6. Ownership

Only the owner[4] of a file can change its permissions. To find out how to change the ownership or group ownership of files, refer to *Using the Network*.

2.7. Modification Time

The modification time indicates the most recent time that the file has been edited, or appended to. You can change a file's modification time, without affecting its contents, with the `touch` command.

```
touch filename
```

Touch does not alter the contents of *filename*, but rather, resets the modification time to the current date and time. If the file does not exist already, `touch` creates it.

2.8. Making Links

A *link* is a name associated with a file. UNIX allows several links to a file at any one time. So, the same file can have more than one name. This is useful when you want to get at a file quickly from within different directories. When you create a file, the system makes the first link, or filename, for you. To make an additional link, use the `ln` command.

```
ln oldname newname
```

If you attempt to make a link to a file in a directory that is on a different disk or disk partition than that of *newname*, you will get an error message of the form:

```
newname: Cross-device link
```

In this case, you can use the `-s` option of `ln` to make a *symbolic* link to the file.

```
ln -s oldname newname
```

A symbolic link is an entry in the directory that points to the *name* of another file, rather than the file itself. A symbolic link can be made across devices, and can be made even when *oldname* does not exist. Because a symbolic link refers to another file's name, rather than the file itself, it may be to your advantage to use a symbolic link instead of a regular link when you want to specify an alternate pathname to the same file.

Both regular (hard) and symbolic links allow you to use *newname* instead of *oldname* to gain permitted access to a file. But, neither a regular (hard) link nor a symbolic link changes the ownership, group, or permissions of a file. So, although you can make a link to a file that you can't read, you still won't be able

[4] or the *superuser*, described in Chapter 5

to read its contents, whichever name you use.

2.9. Seeing File Types with `ls -F`

The `–F` option of `ls` appends a character to the end of each filename to indicate what type of file it is, as follows:

Table 2-4 `ls -F` *File Type Indicators*

tag	*type of File*
(*none*)	normal file
/	directory
*	execute access allowed
@	symbolic link

You may find it useful to place an alias in your `.cshrc` so that `ls` is replaced with `ls –F`:

```
alias ls 'ls -F'
```

2.10. Encrypting Files

You can use `crypt`[5] to encode the contents of confidential files. To encode a file named `secret.plans`, use the following command:

```
mars% crypt < secret.plans > crypt.plans
```

The *angle brackets* are required. The > should be familiar to you. The < is explained in Chapter 3.

Remember to remove the unencrypted version, or your secrets may not keep!

`crypt` then asks you for an encryption *key*. This key is necessary for `crypt` to do its work, and like your password, you must remember it if you want to read your file once again.

```
Key:
```

You can also use `crypt` to decode a file:

```
mars% crypt < crypt.plans > decoy.plans
Key:
```

`decoy.plans` will contain the text you started out with.

If you want to look at the decoded contents, a command of the form:

crypt < *cryptfile* | more

will, after asking for the key, display them on the screen.

You can edit the contents of an encrypted file using the –x option of `vi`.

[5] UNIX encryption facilities are only available to customers within the United States of America.

```
mars% vi -x crypt.plans
Enter key:
```

Whenever you issue the w, or *write*, command, vi runs the file through crypt.

2.11. Searching Through a File with more

There are times when you need to look up something in a long file, but grep won't do because you need to see a whole paragraph or screenful of information, rather than just one line. If the file is very long, stepping through it a screenful at a time with more may take too much time. So, more allows you to search for a string within a file. Instead of typing a [SPACE] to see the next page, or a [RETURN] to see the next line, you can type in a slash (/), followed by a *string*, and more will skip ahead to a screenful containing *string*.

```
mars% more decoy.plans
...
...more 5%...
/picnic
Skipping ...

...
up to the cabin, where we will
have a picnic lunch.
Afterward we could take a swim, and then sip
some sangria.
...
...more 85%...
```

To skip to the next occurrence of that same string, use n.

When using more to look at several files, the command :n will skip to the next file.

2.12. Using pushd, popd and dirs to Change Directories

Sometimes, when you are traveling through a variety of directories, you may find that you want to backtrack. Of course, cd, doesn't remember where you've been. So, unless *you* do, backtracking can be painful. pushd, popd and dirs allow you to stack up a list of directories to revisit.[6] When you are in a directory you'll want to return to, rather than using cd, you can use the

pushd *directory*

command to change directories. Unlike cd, you must specify a *directory*, even when changing to your home directory. pushd changes to the new *directory*, while keeping track of the directory you changed from and to.

If you want to jump back to a previous directory, you can use the

[6] These commands only work with the C-Shell. Refer to Chapter 4, *The C-Shell*, for more information.

```
popd
```

command to work your way back.

If you want to see the list of directories you've stacked up, the

```
dirs
```

command will show it to you:

```
mars% pushd ~
~ ~/env
mars% pushd wwu
~/wwu ~ ~/env
mars% (LBdirs
~/wwu ~ ~/env
mars% popd
~ ~/wwu
```

`dirs`, with the `-l` option, displays the full pathnames stacked directories:

```
mars% dirs -l
/usr/sam /usr/sam/wwu
```

3

More About Commands

3 More About Commands

3.1. Redirecting Output, Redirecting Input, and Pipes

Commands perform actions, typically on data contained in a file. Unless you indicate otherwise, they normally display their results on the terminal screen. The terminal is known as the command's *standard output*.

Because UNIX commands treat files and devices in a uniform way, you can direct the output of a command to any file or device that you choose.

Unless you indicate otherwise, commands normally operate on data as you type it in from the keyboard. So, the terminal is known as the command's *standard input*. Finally, you can use the output of one command as direct input to another, using a special connection symbol called a *pipe*.

Redirecting Output

As you learned in *Getting Started With Unix*, a right *angle-bracket* (>)[7] on the command line indicates that the next word is the name of a file or device in which to place, or *redirect* the output of a command. For instance, the command line:[8]

```
mars% ls -la > list
```

places the output of the `ls -la` command (a detailed list of all files, including hidden files) in a file named `list`.

CAUTION **If a file by that name already exists, any previous contents are deleted *before* the command is performed.**

So, the command

```
cat will.be.empty > will.be.empty
```

removes all existing contents from the file `will.be.empty`.

To avoid writing over existing files, add a line with the command

```
set noclobber
```

to your `.cshrc` file if one isn't there already.[9] Then type in the command:[10]

[7] may be pronounced as "into"

[9] Refer to *Setting Up the UNIX Environment* for more information about this file.

```
mars% source .cshrc
mars%
```

When you are certain that you want to overwrite the previous contents of a file, using a `>!` overrides this file protection.

You can *append*, or 'add to the end of' a file using a *double-right-angle-bracket* (>>).[11] Thus, the command[12]

```
mars% ls >> list
```

adds a second version of output from `ls` (containing just the names of nonhidden files) onto the end of `list`.

Redirecting Input

Just as you can redirect the output of a command, you can also specify a file (or device) from which that command obtains its *input*.

You can use a *left* angle-bracket (<)[13] to redirect the standard input of a command. For instance, the following command prints the contents of the file `list`.

added using `ls -la >`

added using `ls >>`

```
mars% cat < list
drwxr-xr-x  3 sam     512 Jul 29 23:11 ./
drwxrwxrwx  4 sam     512 Jul 19 12:17 ../
drwxrwxrwx  2 sam     512 Jul 26 18:52 SCCS/
-rw-r--r--  1 sam   77293 Jun 27 15:36 csh.1
-r--r--r--  1 sam   21773 Jul 24 16:43 files
-rw-r--r--  1 sam       0 Jul 29 23:11 list
lrwxrwxrwx  1 sam       8 Jul  8 16:40 outline -> ../wwu.b
-rw-r--r--  1 sam    3557 Jul 12 18:59 philos
-rw-r--r--  1 sam      82 Jul 24 16:43 pic.src
-r--r--r--  1 sam    1381 Jul 12 15:50 preface
SCCS/
csh.1
files
list
outline@
philos
pic.src
preface
```

[10] If using windows, type this `source` command in each `shelltool` or `cmdtool` window, so that the change will take effect in the C-Shell running within each.

[11] may be pronounced as "onto"

[12] With `noclobber` set, a file must already exist before the standard output can be appended to it. Using a `>>!` overrides this.

[13] may be pronounced as "from"

Most commands allow the input file to be specified as an argument. You could, for example, produce the same display with the command:

```
mars% cat list
```

However, other commands, such as `crypt`, require use of <, the input redirection symbol.

Pipes and Pipelines

The output of one command can be fed in directly as input to another. A set of commands strung together in this way is called a *pipeline*, and the symbol for this input/output (I/O) connection is a vertical bar (|),[14] called a *pipe*. Pipes and pipelines have a wide variety of uses.

For example, suppose you wanted only to list symbolic links in the directory. You can combine `ls` and `grep` to get the result you want. The pipeline

```
ls -l | grep lrwx
```

will do the trick, as will the pipeline

```
ls -F | grep @
```

A less efficient way to accomplish would be:

```
ls -l >filename
grep lrwx <filename
rm filename
```

There is no filename following `grep` because the pipe symbol indicates that `grep` is to search through its standard input, which in this case is the output of `ls`.

You can connect several commands to make longer pipelines. For instance, the command line:

```
mars% ls -l | grep lrwx | wc
       1      10      65
```

uses `wc` (word count) to display the number of lines, words, and characters, respectively, in the list of symbolic links culled from the output of `ls` by `grep`. Since `wc` received only one line from `grep`, there was only one symbolic link in the directory.

The ability to 'cook up' intricate commands on the spot is a very special feature of the UNIX system, and one that becomes increasingly useful as you continue to experiment and learn.

Filters

Commands like `grep` are called *filters*. They accept text as input, transform it in a straightforward way, and produce text as output. Although often used as commands in their own right, filters are especially useful in pipelines.

`ls` is not a filter, because it doesn't accept data from the standard input. Neither is `date`. As you might expect, the command `ls | date`

[14] may be pronounced as "through"

produces *only* the date, since `date` ignores its standard input. What does `date | ls` produce?

`more` is another type of filter. It transforms the data by breaking it up into screen-sized chunks. Some other interesting filters are:

`head -`*n*	displays the first *n* lines of a file. With no -*n* argument, it displays the first ten lines.
`tail -`*n*	displays the last *n* lines. With no -*n* argument, it displays the last ten.
`tail +`*n*	skips to line *n* and displays that line through the end of the file.
`more "+/`*pattern*`"`	like `tail`, this command begins printing two lines before the first match for *pattern*, which can be either a string or a `grep` search pattern (described below under `grep` *and* `grep` *Search Patterns*).
`cat -v`	translates nonprinting characters into strings of regular characters of the form `^c` (for control characters), or `M-`*c* (for 8-bit characters).
`sort`	display the line in alphanumeric order, or according to an order you specify. Refer to `sort` in the *Commands Reference Manual* for more information.
`sort -n`	sort in numerical order.
`fmt`	does rudimentary formatting of text.
`rev`	reverses the order of characters within each line.
`pr -t -`*n*	breaks up the output into *n* columns. The `-t` option suppresses a heading that would otherwise appear.
`spell`	produces a list of possibly-misspelled words.
`sed`	performs simple edits on a line-by-line basis. For instance, the alias: `alias grep 'grep \!* | sed "s/:/:   /"'` Improves the appearance of `grep` output by substituting a ``colon-plus-three-spaces'' for the first ``colon'' on a line (if any). Compare:

The command

`look` *string*

looks up words (in the system dictionary) whose leftmost characters match *string*. The command

`look a`

will display all words starting with `a`. To further restrict the seach, add more characters.

```
mars% grep "H C" *
c.shell:.H C "The C-Shell"
commands:.H C "More About Commands"
files:.H C "More About Files"
intro:.H C "Introduction"
manag:.H C "Managing Your Files"
preface:.UH C "Preface"
printr:.H C "More About Printing"
proc:.H C "Processes and Other Users"
```

with:

```
mars% alias grep 'grep \!* | sed "s/:/:   /"'
mars% grep "H C" *
c.shell:   .H C "The C-Shell"
commands:   .H C "More About Commands"
files:   .H C "More About Files"
intro:   .H C "Introduction"
manag:   .H C "Managing Your Files"
preface:   .UH C "Preface"
printr:   .H C "More About Printing"
proc:   .H C "Processes and Other Users"
```

Or, you could get fancy and use a (TAB) rather than three spaces for better alignment. Refer to *Using UNIX Text Utilities* for more information about sed.

Example of Filters in Action

One clever trick is to create a rhyming dictionary of words using filters and the system dictionary:

```
mars% rev /usr/dict/words | sort | rev | pr -t -3 | more
St.             UK              Elba
NCAA            BTL             alba
FAA             TTL             samba
NOAA            SIAM            marimba
ABA             IBM             Zomba
MBA             ACM             Manitoba
YMCA            CACM            Cuba
RCA             JACM            Hecuba
YWCA            SCM             scuba
FDA             FM              Aruba
ERDA            GM              tuba
USDA            NM              catawba
CIA             PM              Ithaca
USIA            RPM             portulaca
UCLA            ASTM            Dacca
AMA             CERN            Decca
BEMA            USN             Mecca
...more...
```

As noted above, rev reverses the character order of each word. Since each word appears on a line by itself in the system dictionary, rev reverses the order of characters in each word. sort then sorts the words in order of (what was) their last character. A second pass through rev reverses the characters in each word a second time so that they read correctly, and you have the makings of a rhyming dictionary! Piping this through pr and more, yields a more readable display.

Using the tee Command

Suppose that you want to send duplicate output both to the terminal screen, and to a file for future reference. When placed in a pipeline, the tee command lets you direct output to more than one destination. For example, the pipeline

```
mars% ls -l | grep lrwx | tee newlist
```

displays the list of symbolic links on the screen and creates a file `newlist` that contains a copy of this information as well.

With the `-a` option, `tee` appends the data onto named files that already exist. So the command:

```
mars% ls -l | grep lrwx | tee -a newlist
```

adds this information to `newlist` once again (displaying it on your screed as well).

Redirecting the Standard Error

When a command performs without problems, it produces results on its standard output. When that command encounters a problem, however, it uses a different channel to send error messages, or *diagnostic output*, to the terminal. This second channel, called the *standard error,* can also be redirected.

You can redirect the standard error to the same destination as the standard output by appending an ampersand (`&`) to the output redirection symbol.

`>&` sends both standard and diagnostic output to a destination file.[15] `>>&` appends the output to the file. `|&` includes both types of output as input to the next command in the pipeline.

If you want a command to perform silently, that is, to display no output of either kind, you can redirect its output to `/dev/null`, the system ''wastebasket.''

command `>& /dev/null`

To separate the standard error from the standard output, use a command line of the form:[16]

`(`*command* `>` *outfile*`) >&` *errorfile*

When you want to force output to appear on the terminal, you can redirect it to `/dev/tty`, (a synonym for) the name of the terminal.

command `>& /dev/tty`

So, the command

```
mars% (nroff /usr/dict/words > /dev/null ) >& /dev/tty
```

throws away any formatted output and displays only the error messages produced by `nroff` (if any). This construction can save you time when testing long-

[15] The Bourne shell uses the symbols: `2>&1` to accomplish this.

[16] In the Bourne shell:
command `>` *outfile* `2>` *errorfile*

running commands.

3.2. Escape Character, Quotes, Separation and Continuation Symbols

To indicate that a special character or symbol is to be taken as literal text, precede it with a backslash (\). By prepending the backslash, you *escape* the special meaning of the symbol.

You can use double quotes (") to surround text that you want to be interpreted as one word. You can also use single quote marks (') to surround text that you want to be interpreted literally (no filename substitution, for instance).[17] In either case, you may still need to use a backslash to treat symbols (such as &, !, $, ?, and \) within the string as ordinary characters.

To place more than one command on a single command line, separate them with a semicolon (;). For instance:

The echo command simply repeats its arguments on its standard output.

```
mars% echo The Scarlet Letter > tempfile ; rm tempfile
```

puts the words The Scarlet Letter into tempfile, and then removes that file. To continue a command onto the next line, use a backslash to escape the RETURN key.

```
mars% rev /usr/dict/words | \
sort | rev > rhymes
```

produces the rhyming dictionary described above. The terminal displays the carriage return, but the system ignores it.

3.3. grep and grep Search Patterns

You can use grep to search for *patterns* much like those you are familiar with from *Filename Substitution*.

Although the action is similar to that of filename substitution, the way you specify search patterns is different. Because they search through lines of text, grep search patterns, or *regular expressions*[18] cover a broader range of text patterns than those for filename substitution, and they have a different *syntax*.[19] Some characters with special meaning to grep also have special meaning to the system and need to be quoted or escaped. So, whenever you use a grep regular expression on the command line, surround it with quotes, or escape such characters as &, !, ., *, $, ?, and especially \, with a backslash.

Within a regular expression, dot (.) matches any single character (like ? in filename substitution). So the command,

[17] Within single quotes, neither filename substitution, nor other forms of substitution to be described in Chapter 4, are applied.

[18] The name grep is derived from the ed search and print command: g/*regular-eexpression*/p

[19] Although not a formal definition, you can think of the *syntax* of a command or argument as a rule for typing it in correctly.

```
mars% grep '.b' list
```

matches all lines in which b is preceded by a character. In effect, this matches all lines containing b, except when b is the first character on the line.

A caret (^) anchors the pattern to the beginning of the line. So the command

```
mars% grep '^b' list
```

matches any line starting with b. A dollar-sign ($)
anchors the pattern to the end of the line. The command

```
mars grep '^b$' list
```

matches any line in which b is the only character.

Bracketed lists and ranges work just as they do for filename substitution, but the asterisk (*) doesn't. When the asterisk follows a character, grep interprets it as 'zero or more instances of that character'. When the asterisk follows a regular expression, grep interprets it as 'zero or more instances of characters matching the pattern'. To match zero or more occurrences of any character, use

```
.*
```

Suppose you want to find lines in the text that have a period in them. Preceding the dot in the regular expression with a backslash (\) tells grep to ignore (*escape*) its special meaning. The expression

```
^\.
```

matches lines starting with a period, and is especially useful when searching for nroff formatting requests.

Table 3-1 grep *Search Pattern Elements*

character	*matches:*
^	The beginning of a text line.
$	The end of a text line.
.	Any single character (like ? in filename substitution).
[...]	Any single character in the bracketed list or range.
[^...]	Any character not in the list or range.
*	Zero or more occurrences of the *preceding character* or *regular expression*. (Not like filename substitution.)
.*	Zero or more occurrences of any single character. Equivalent to '*' in filename substitution.
\	Escapes special meaning of next character.

Going back to the rhyming dictionary, we can now use grep to produce an alliterative list of rhyming words starting with a:

```
mars% rev /usr/dict/words | sort | rev | grep "^a" \
| pr -t -3 | more
a              anthropomorphic   apocalyptic
amoeba         anorthic          antagonistic
alba           acyclic           anachronistic
armada         angelic           autistic
addenda        alcoholic         atavistic
agenda         apostolic         agnostic
anaconda       acrylic           acoustic
althea         aerodynamic       attic
azalea         academic          aeronautic
area           algorithmic       astronautic
alfalfa        astronomic        analytic
alga           autonomic         arc
...more...
```

Refer to `grep` in the *Commands Reference Manual* for more information about regular expressions and the `grep` family of commands.

4

The C-Shell

4

The C-Shell

4.1. Overview

When you type in a command, you can expect certain things to happen. By now you know that if you misspell a command the system replies with an error message. You then get a new prompt so that you can try again. When you type in the command correctly, the system waits for it to finish before giving you another prompt (unless you put it in the background with an `&`).

Of course, these things don't just happen by magic. A program, called a *shell* accepts and interprets what you type, passes your interpreted commands on to be performed, and waits for each to finish before proceding to the next.

Although the shell waits before issuing a prompt, the terminal allows you to type ahead. That is, the terminal displays what you type and passes each line along when the shell (or interactive program like `vi`) is ready for it.

There are two shells available on the Sun Workstation, the C-Shell, and the Bourne shell. The C-Shell has convenient features for interactive use, and we assume that you are using it for this purpose. The Bourne shell has fewer conveniences, but runs faster, and has a simpler syntax for writing command routines, called *scripts*.

The system starts a shell whenever you log in or create a terminal with `shelltool`. Technically speaking, the *C-Shell* is known as a *command interpreter*. You can think of the C-Shell as a layer of software between you and the system's internal workings.

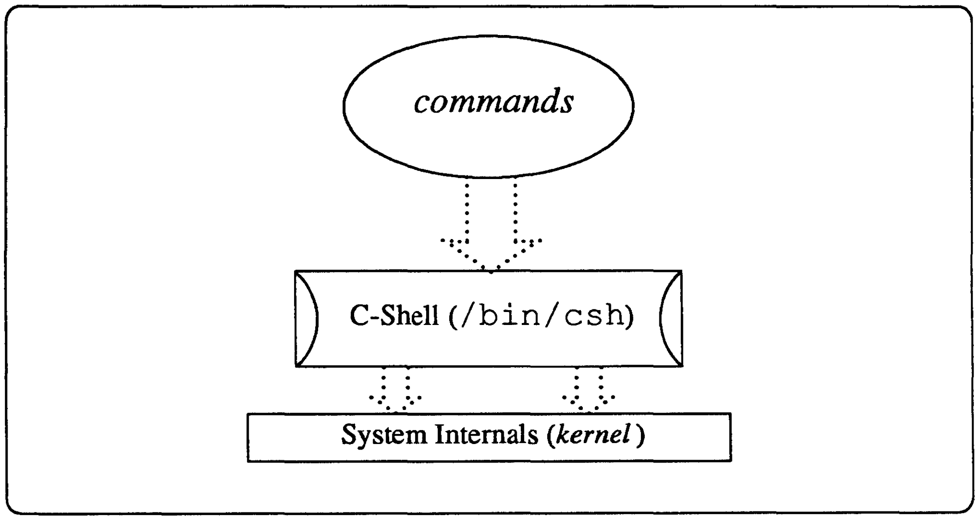


Figure 4-1 *The C-Shell and Commands*

Filename substitution is one example of how the C-Shell interprets what you type. When you use the * wild card, the C-Shell compares it against entries in the directory and builds a list of filenames that match. It then replaces the wild card with the list, sending this expanded version of the command you typed on to the control of the system's internal scheduling mechanisms.

The way the C-Shell performs *alias substitution* is another example. When you type in an alias, the C-Shell recognized it as such, and replaces it with the more complex command or, *expansion* that you have assigned to it.

A *shell* is an interactive program just as are `Mail` and `vi`. You can switch to a new C-Shell, just as you can switch to `vi` by typing in the `csh` command. To escape such a *subshell* use CTRL-D or `exit`.

You can run a command within a *noninteractive* C-Shell by placing it within parentheses on the command line. You have already seen an example of this in *More About Commands*, where a *subshell* is used to separate the standard output from the standard error:

```
( command > outfile ) >& errorfile
```

The C-Shell provides features that you can use to further simplify entering of commands. In addition to repeating previous commands, you can use the `history` mechanism to modify them. You can put "placeholders" within alias definitions to simplify complicated commands and pipelines. And, you can define *variables* to stand for long strings or lists of words.

These and other features make the C-Shell easy to work with and easy to customize.

4.2. History Substitution and Command-Line Editing

The C-Shell keeps a list of previous commands that you have typed in. The `history` variable determines the length of this list.

Add this command to your `.cshrc` file if it isn't already there.

To set or change this variable, use a command of the form:

```
set history=n
```

where *n* is the number of commands to remember.

Reviewing Commands

To see the list of previous *events*, or command lines, type `history` after the prompt.

```
mars% history
   1  ls
   2  cd
   3  grep -v done tasklist
   4  history
```

Repeating Commands

As you learned in *Getting Started With UNIX*, you can repeat the most recent event by typing in two exclamation points (`!!`). The history mechanism lets you repeat any command in the events list by typing an exclamation point, followed by its command line number,

`!`*n*

for example:

```
mars% !3
grep -v done tasklist
...
```

You can specify the *n*'th command back,

`!-`*n*

as in:

```
mars% !-3
cd
```

You can repeat an event by typing an exclamation point, followed by the first few characters that match it,

`!`*str*

The history mechanism performs the first match it encounters. You may have to add a few characters to get the desired event. In this example,

```
mars% history
...
11   cd
12   ls -l old
13   ln -s old/stuff new
14   history
mars% !l
ln -s old/stuff new
^C
```

Because the user typed in too few characters to specify the event precisely, `!l` matched the most recent event beginning with `l`, namely `ln`, (even though this wasn't the event desired). The observant user interrupts it, and then types in `!ls` to match the desired event:

```
mars% !ls
ls -l old
...
```

Sometimes it's easier to match against a string of characters *embedded* within the the event. To repeat a command in this way, use:

!?*str*?

where *str* is the embedded string to search for. For example:

```
mars% !?stuff?
ln -s old/stuff new
```

Command Line Editing

A word on the command line that begins with an exclamation is referred to as an *event designator*. An event designator can stand for a previous command, or selected words from a previous command line.

You have already seen how to edit the previous command using quick substitution (^*old*^*new*^). And, you have seen how to repeat the last word of the previous command (!$). The history mechanism provides you with the means to select any word from any event in the history list, and to modify it. In some cases, it can be easier just to type the new command directly. But in many cases, command line editing can save you time and keystrokes.

You can place a :p on the end of an event designator or quick substitution to prevent the expanded command from being performed. The shell interprets the command, echos it, and places it in the history list. This gives you a chance to look at the expanded version before actually running it. If it checks out, you can use !! to run it. Otherwise you can do successive edits using

^*old*^*new*^:p

until you get it just right.

Suppose that you want to apply several commands to a long list of files, and you don't want to have to retype the list every time. !* repeats all arguments to the previous command (all but the first word of the command line). !^ expands to the first argument. If the last command was

```
echo first
```

!^ would expand to `first`. !:*n* expands to the *n*'th argument (*n*+1'th word).

Selecting Words Within Events

You can select a specific word from a specific event by appending a *word designator* to its event designator. A word designator has the form of a colon, followed by a character. :* expands to all arguments in the event. Using the history list above,

```
grep !?stuff?:*
```

expands to,

```
grep -s old/stuff new
```

a command that doesn't say very much when it works.

`:$` expands to the last argument of the selected event. `:^` expands to the first argument. `:`*n* expands to the *n*'th argument.

Modifying Selected Words and Events

You can edit the text of an event or word by appending an *event modifier* to it. A modifier starts with a colon, followed by one or more characters that indicate the actions to perform. `:s/`*old*`/`*new*`/` substitutes *new* for *old* in the first word where there is a match for *old*. When inserted between the colon and the modifier, a `g` indicates that the modifier applies to all designated words. So,

```
grep !?stuff?:*:gs/s/N/
```

expands to

```
grep -N old/Ntuff new
```

which results in a scan for the string `-N`, a *'file not found'* message, and a list of occurrences of `-N` in the file '`new`'.

As mentioned above, `:p` indicates that the event or word is to be expanded and echoed, but not performed. You can place several modifiers in an event or word designator. For instance:

```
grep !?Ntuff?:*:gs/N/S/:p
```

is echoed as

```
grep -S old/Stuff new
```

but not performed.

For more information about event designators, word designators, and event modifiers, refer to Appendix D, *C-Shell Special Characters*.

4.3. Amazing Aliases

You can use *escaped* event and word designators within alias definitions to create aliases for complicated commands and pipelines. When you use the alias as a command, the escaped event designator (such as `\!*`) is replaced by command line arguments that you then type in. For instance, you might want to create an alias for a pipeline to format and then print a file.

An `alias` for `nroff` with the proper options is easy, because no characters follow the arguments you supply when using it:

```
mars% alias format 'nroff -ms'
mars% format file1 file2

formatted text appears
```

But, if you want to get the the formatted output to the printer with the same command, you must supply a pipe symbol, followed by `lpr`. Rather than having to type these characters in every time, you can use the event designator `\!*` within the definition to stand for all arguments to `nroff`. When you actually run the command, the C-Shell replaces the placeholder with any words that follow `print` on the command line.

```
mars% alias print 'nroff -ms \!* | lpr &'
mars% print file1 file2
[1] 2832
printed output comes out of the printer later on
```

This alias has the added benefit of running both `nroff` and `lpr` in the background.

You can also use the command-separation symbol `;` to create aliases that perform several commands in succession.

An event designator can be used more than once within an `alias` definition.

```
mars% alias rw 'chmod +rw \!* ; ls -l \!*'
mars% rw file1 file2
-rw-rw-rw-  1 user       1699 Jul 23 13:32 file1
-rw-rw-rw-  1 user       1023 Jul 20 10:18 file2
```

Another alias that is quite useful tells you which directory you've changed to whenever you use `cd`:[20]

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

Escaping an Alias

To run the unaliased version of a command, precede the name of that command with a backslash:

```
mars% rm test
rm: remove test? ^C
mars% alias rm
rm -i
mars% \rm test
mars%
```

4.4. Variable Substitution

A *variable* is a named location in which to store text that you'd like the C-Shell to remember for you. You can use the `set` command to associate a variable name with a word to remember. A placeholder, composed of a dollar-sign (`$`), followed by the name of a variable, is replaced with the contents of that variable by the C-Shell. Thus, you can use a variable name, preceded by a `$`, as an abbreviation for its contents.

To assign a value to a variable, type in a command like:

```
mars% set testdir = ~/programs/test
```

[20] Although you could use `\!:1` instead of `\!:*` (since `cd` gives an error message when used with more than one argument), it is simpler to figure out what is going on if your aliases preserve, as closely as possible, the original behavior of commands they replace.

To display that variable's contents:

```
mars% echo $testdir
~/programs/test
```

Suppose that you are working with files in two directories, each with very long, and very different pathnames:

```
/usr2/sam/sources/gfx/lines/module3
/usr/bin/c/gfx/lines/module3
```

You can abbreviate these pathnames as follows:

```
set src = /usr2/sam/sources/gfx/lines/module3
set bin = /usr/bin/c/gfx/lines/module3
```

Then, when you want to perform commands on files in these directories, you can use `$src` instead of `/usr2/sam/sources/gfx/lines/module3`, and `$bin` instead of `/usr/bin/c/gfx/lines/module3` on the command line:

```
mars% cd $bin;pwd
/usr/bin/c/gfx/lines/module3
mars% cd $src;pwd
/usr2/sam/sources/gfx/lines/module3
```

The `set` command with no arguments prints a list of all C-Shell variables and their current values. To see the value of a single variable, use a command of the form:

echo $*variable*

Storing Lists in C-Shell Variables

In addition to single words, you can store a list of words in a C-Shell variable by enclosing the list in parentheses when you use the `set` command. One example of this is the `path` variable that you set in your `.cshrc` file. Another might be:

These directories contain source files, and formatted versions, respectively, of Section 1 of the online Manual Pages.

```
mars% set mdirs = (/usr/man/man1 /usr/nan/cat1)
mars% echo $mdirs
/usr/man/man1 /usr/cat/man1
```

You can select a specific word from the list by appending an *index* to the *call*[21] to the variable as follows:

$*var*[*n*]

where *var* is the name of the variable, and *n* is a number indicating the position of the word within the list. Using the above example, the word

[21] A call to a variable is the string you use to indicate that what you really want is the value it contains, in this case the name of the variable preceded by a dollar-sign.

`/usr/man/cat1` is the second word in the list. So, the command:

```
echo $mdirs[2]
```

displays the value

```
/usr/man/cat1
```

You can also specify a range:

```
mars% echo $mdirs[1-2]
/usr/man/man1 /usr/man/man2
mars%
```

But, if you enclose a number in the braces that is higher than the count of words in the variable, you will get an error message. You can use filename substitution to simplify entering a list. The command:

```
set man = (/usr/man/{man,cat}?)
```

yields the following value:

```
mars% echo $man
/usr/man/man1 /usr/man/man2 /usr/man/man3 /usr/man/man4
/usr/man/man5 /usr/man/man6 /usr/man/man7 /usr/man/man8
/usr/man/cat1 /usr/man/cat2 /usr/man/cat3 /usr/man/cat4
/usr/man/cat5 /usr/man/cat6 /usr/man/cat7 /usr/man/cat8
```

which is a complete list of all the directories containing Manual Page sources and formatted files.

Processing Lists with `foreach`

The `foreach` command provides a means to apply a set of commands successively for every word in a list. It prompts you for a set of commands, uses an *index* variable to store the current word while executing each pass through the commands, and repeats the list of commands once for each word in the list.

The syntax of the `foreach` command is:

`foreach` *index* (*list*)

where *index* is the name of the variable, and *list* is a list of words. After you type in the [RETURN], `foreach` prompts for a command with a question mark. It continues to prompt for commands until you type the command `end` by itself after the question mark. This signifies the end of the loop.[22] For instance:

```
mars% foreach file (*)
? echo -n $file
? echo -n ", "
? end
```

[22] A *loop* is a set of commands to repeated successively.

yields a new variation on a very familiar theme, the list of files:

```
... c.shell, commands, csh.blt, csh.var, ...
```

You can use variable substitution, as well as filename substitution symbols within the list.[23] Using the variable `man` defined above, the following `foreach` loop gives you a count of the source files and then the formatted files within each section of the Manual Pages. As the loop proceeds, the value of the index variable (written as `$dir`) changes with each pass.

```
mars% foreach dir ($man)
? echo -n $dir
? ls $dir | wc -l
? end
/usr/man/man1     264
/usr/man/man2     118
/usr/man/man3     155
/usr/man/man4      47
/usr/man/man5      49
/usr/man/man6      36
/usr/man/man7       8
/usr/man/man8     108
/usr/man/cat1     264
/usr/man/cat2      94
/usr/man/cat3     154
/usr/man/cat4      47
/usr/man/cat5      49
/usr/man/cat6      36
/usr/man/cat7       8
/usr/man/cat8     108
```

Predefined Variables

The C-Shell maintains a set of predefined variables. Some of these, like `noclobber`, are used by the C-Shell to affect the way it behaves. Others keep track of information that the C-Shell needs to know about. `home`, for instance, keeps a record of your home directory. If you change the value of `home`, and then use `cd` with no argument, the C-Shell attempts to change directories to that new value.

```
mars% set home=/
mars% cd;pwd
mars% set home=nonesuch
mars% cd;pwd
cd: Can't change to home directory.
mars% echo $home
nonesuch
mars% cd ~
nonesuch: No such file or directory
```

[23] This also works with the `set` command.

For a complete list of C-Shell predefined variables and their uses, refer to Appendix E, *C-Shell Predefined Variables*.

Environment Variables

The C-Shell also maintains a set of variables, called *environment* variables. You should be familiar with them from reading *Setting Up the UNIX Environment: Beginner's Guide*. Environment variables are passed along to any commands or subshells. They are created and modified using the `setenv` command, which has a different syntax than that of `set`.

```
setenv name value
```

There is no equal sign between the name of the variable and its value, as there is with `set`. And, only one word (or string within quotes) can be assigned to an environment variable.

Environment variables are passed to all commands and programs run from within the current shell. C-Shell variables are only effective within the *current* shell.

Typically, the names of environment variables are given in all capitals. In some cases, there is a lower-case equivalent used by the C-Shell.

Others include: `user` and `USER`, `term` and `TERM`, `shell` and `SHELL`, and `path` and `PATH`

The environment variable `HOME` is such a case. When you use the `set` command to change the value of the (`home`) shell variable, the equivalent environment variable is also changed. When you use `setenv` to change the environment variable, however, the value of the `home` shell variable is not affected:

```
mars% set home=bogus
mars% echo $home
bogus
mars% echo $HOME
bogus
mars% setenv HOME /usr2/sam
mars% echo $home
bogus
mars% echo $HOME
/usr2/sam
mars% set home=/usr2/sam
```

To get a list of all environment variable and their current values, use the command `printenv`.

4.5. Command Substitution

The term *command substitution* is a bit misleading. A better term would be *output* substitution, because it allows you to use the output of other commands as arguments on the command line.

When you surround a command with backquotes (`) anywhere on the command line the C-Shell starts a subshell, executes the command within the subshell, and substitutes the resulting output for the backquoted text.

`echo` is a useful command for testing the results of filename, variable, and command substitution.

```
mars% echo `ls -l | head -1`
total 20
mars% ^-l^^
echo `ls | head -1`
News
mars% ^echo^chmod 775^
chmod 775 `ls | head -1`
```

4.6. Job Control

UNIX is a *multitasking* operating system. This means that it can keep track of several users and their commands simultaneously. The system also allows you to run several commands at once by placing them in the background. The C-Shell provides you with the means to inquire about, stop, or bring to the foreground any job started through it.

Because each window runs with a different shell, you can't use job control to inquire about jobs started from different windows.

To see how job control works, start a background job that won't finish until you tell it to:

```
mars% vi test &
[1] 4001
```

The [1] is the *job* number. The 4001 is a *process number* that you can ignore for now.[24] In this case, number 1, running vi, is the only job that is either stopped or running in the background. When vi attempts to write its startup message to the terminal, it does not succeed because control of the terminal belongs to the C-Shell. So, vi stops, and waits for you to give it access to the terminal. The C-Shell reports any change in the status of jobs under its control, so you see a message that looks like:

```
[1] + Stopped (tty output) vi test
```

when the C-Shell issues the next prompt. Notice the plus sign. This indicates that the job is *current*, meaning that it is the most recent job to have stopped. A minus sign indicates that a job is *next*. When the current job is finished, a job so marked will become current.

To give a job access to the terminal, or 'bring it into the *foreground*', type in

%*n*

where *n* is the job number. If you omit the job number, the C-Shell brings the current job forward. When you stop an interactive program like vi, it waits, under job control, for you to start it running again. So, if you want to stop in the middle of vi without losing your place, you can type a CTRL-Z. vi stops, and the C-Shell resumes control of the terminal until you type in a %.

```
mars% %1
the same vi screen comes up
```

[24] Processes are described in Chapter 5, *Processes and Other Users*.

To stop the job once again, type in a CTRL-Z.

```
vi screen
^Z
Stopped
mars%
```

Stopping a job and resuming it can be useful when you have large programs (such as `nroff`) running, and you need to do something quickly. Rather than opening a new `shelltool` or `cmdtool`, or waiting for the big program to finish, you can stop (or *suspend*) it temporarily, perform your urgent task, and then resume the big program from where it left off.

To see what jobs are either stopped or running in the background, type in `jobs`.

To indicate that a stopped job should continue to run in the background, type in

`%n &`

```
mars% nroff -ms hugefile vastfile | lpr
^Z
Stopped
mars% jobs
[1] - Stopped (tty output) vi test
[2] + Stopped              nroff -ms hugefile vastfile
mars% %2 &
[2]   nroff -ms hugefile vastfile | lpr &
mars%
```

To abort a background job, use a command of the form:

`kill %job`

where *job* is the number of the job to kill.

```
mars% kill %1
[1]   Terminated        vi test
```

Exiting With Stopped Jobs

If you try to exit a shell while a job is stopped, you get the warning message:

```
There are stopped jobs.
```

A second `logout` will then log you out (but its a good idea to see what's back there with `jobs` before you exit).

5

Processes and Other Users

5

Processes and Other Users

5.1. Processes

After each command is interpreted by the C-Shell, UNIX creates an independent *process*, with a unique process ID number (PID), to perform it.[25]

The system juggles its time and *resources* amongst the various processes currently running, and uses the PID to track the progress, current status, the amount of time and the percentage of available memory each process uses.

The C-Shell passes its environment variables[26] (created by the `setenv` command) and their values along to the processes it starts. These are known as *child* processes. A child process may also create new children of its own.[27] In general, when a process creates a child, it waits for the child to finish before proceeding with its own tasks. As each child process completes its work, it sends an exit status number, or *return code* to its parent process. Most programs that finish normally exit with a return code of `0`. Programs that encounter errors typically exit with a status of `1` (or some other number).

To see what processes you have running, use the `ps` command. In addition to showing the PID for each process you own (created as a result of a command you typed in), `ps` also shows you the terminal from it was started, its current status (or *state*), the cpu time it has used so far, and the command it is performing.

```
mars% ps
  PID TT STAT  TIME COMMAND
 2649 co IW    0:23 suntools
 2650 p0 IW    1:12 shelltool -C
 2651 p0 IW    0:06 -bin/csh (csh)
 6006 p1 R     0:02 ps
 2655 p2 S    34:32 shelltool
 2659 p2 IW    0:50 -bin/csh (csh)
 6000 p2 R     0:05 vi proc
```

The table below should help decipher the display.

[25] Technically speaking, a process is an area in memory that contains a copy of the *program* indicated by the command you typed in, along with any data from the files you supplied as arguments (or from your terminal).

[26] It does not pass along shell variables (created by `set`).

[27] The parent is said to `fork` a child process.

Table 5-1 *Information Displayed By* `ps`

Column	Symbol	Meaning
PID		process ID number
TT		terminal:
	co	/dev/console
	mn	/dev/tty*mn*
STAT		state of the process:
	R	runnable (running)
	T	stopped
	P	paging
	D	waiting on disk
	S	sleeping (less than 20 seconds)
	I	idle (more than 20 seconds)
	Z	terminated, control passing to parent
	W	swapped out[29]
	>	exceeded soft memory limit
	N	priority was reduced
	<	priority was raised
TIME		processing time (so far)
COMMAND		command being performed

Terminating a Process with `kill`

`kill` provides you with a direct way to stop commands that you no longer want, even from a shell running on another terminal or from another window. This is particularly useful when you make a mistake typing in a command that takes a long time to run, such as `troff`.[30]

To terminate a process, type `ps` to find out the process ID.

You can pipe `ps` output through `grep`:
`ps | grep` *command-name*

`When you see which process or processes to terminate, type in kill followed by the PIDs for those processes.`

```
mars%troff -Tlp -ms much.too.big.doc
^Z
Stopped
mars% ps | grep troff
6788 p2 S    34:32 troff -Tlp -ms much.too.big.doc
mars% kill 6788
[1]   Terminated     troff -Tlp -ms much.too.big.doc
mars%
```

Use `kill -9` to forcefully terminate a process.

[29] Of the various states in the `STAT` column, `IW` can be an indication that a process is in trouble. If you find a process in this state, and if in 5 minutes or so it is still in that state, it is probably a good idea to terminate it and run the command again (checking to be sure that the command line makes sense and is typed in correctly).

[30] `troff` is a powerful text formatter that can prepare typeset-quality documents like this one.

`kill` will accept either a PID number, or a job number preceded with a `%` (`%1`, for instance) as an argument. You can, however, set up an alias that will search for a command by name and terminate the first process it finds running that command:[31]

```
alias slay 'set p=`ps|grep \!*|head -1`; echo $p; kill -9 $p[1]'
```

The first part of this alias (up to the semicolon) searches for the command that you supply as an argument, strips off all but the first occurrence and stores the output line in the variable `slay`. The second part displays which process it is about to kill. The third part selects the first word in the variable `slay` (the PID), and kills the process with that number.

```
mars% view &
[1] + Stopped (tty output) view
mars% slay view
1154 p3 T 0:00 view
mars%
```

Timing Processes

To keep track of the system resources used by a particular command, type in `time`, followed by the command:

```
mars% time wc file
58      57     536 file
0.0u 0.2s 0:01 24% 1+1k 6+0io 0pf+0w
mars%
```

`time` displays statistics about the command as follows:

Table 5-2 *Information Displayed By* `time`

Column	*Explanation*
`_._u`	user time
`_._s`	system time
`_:__`	elapsed time
`__%`	cpu time as a percentage of elapsed time
`_+_k`	average shared memory, plus average unshared memory (kilobytes)
`_+_io`	number of block input operations, plus block output operations
`_pf+`	page faults
`_w`	swaps

[31] When you desire functions that are more complex than this, such as performing steps repeatedly or making use of more than one variable, you should consider writing a shell script to perform it. See Appendix F for information about writing Bourne Shell scripts, or Appendix B for information about C-Shell scripts.

When a command runs for longer than a certain number of cpu seconds (determined by the `time` C-Shell variable), these statistics are displayed automatically.

Running a Command at a Later Time with `at`

You can take advantage of hours when the system is not heavily used to run large jobs that require a large amount of system time or memory (like formatting large documents with `troff`).

First, create a file containing the command line you wish to run later on:

```
mars% cat > atfile
troff -ms much.too.large.document
^D
mars%
```

Then type in `at`, followed by the time you wish to run the job, and the name of the file containing the command line(s).

```
mars% at 2a atfile
mars%
```

This command tells the system to start formatting and printing the large document at 2:00am. You can use up to four digits to specify the time in hours and minutes, followed by an `a` for am, or `p` for pm.

5.2. Other Users

By now you've realized that to the system you're not just another pretty face. From the system's standpoint, every user has a login name, an identification number or *userid*, a password, a group membership, a user's name or other pertinent data, a home directory, and a default shell. This information is kept in the file `/etc/passwd`. To find out who can log in to your system, look in this file.[32]

```
mars% more /etc/passwd
root:0XtYHFnkYou3Y:0:10:Operator:/:/bin/csh
daemon:*:1:1::/:
uucp:eXs0qzRjUOS8Y:4:4::/usr/spool/uucppublic:
cindy:Lu8UBYYbPNEpw:26:20:Cindy Smith:/usr2/cyndi:/bin/csh
carter:SQxRMoQbqQ0Hk:612:20:Jamie Carter:/usr2/carter:/bin/csh
jimg:lUvG9UKY0uE/A:1131:60:Julie Gomez:/usr2/jimg:/bin/csh
ben:bAwVM.A6LiXFo:1132:30:Ben Benson:/usr2/ben:/bin/csh
karla:mceurlTqKdcDQ:1172:30:Karla Caracas:/usr2/karla:/bin/csh
---More---
```

Fields corresponding to the above categories are separated by colons, and described in the following table (using the last line above as a sample entry).

Table 5-3 *Information Contained in* `/etc/passwd`

Field	*Sample*
login name	karla
encrypted password	mceur1TqKdcDQ
user ID number	1172
group ID number	30
commentary	Karla Caracas
home directory	/usr2/karla
login shell	/bin/csh

The first line of this file contains an entry for `root`, the operator of the system. When logged in as `root`, the operator can access any file or device on the system, perform system maintenance, and edit system files such as this. The next two entries allow for certain networking functions to be performed, and the subsequent lines correspond to individual users.

Users Currently Logged In

The system tries to provide equivalent performance to everyone using it. To find out who is logged in, type `who`.

```
mars% who
patti    tty07   Aug 29 07:57
alder    tty08   Aug 30 09:08
domke    tty09   Aug 30 08:44
bartlett tty0c   Aug 30 11:35
jd       tty10   Aug 26 11:08
gabe     tty13   Aug 30 05:38
jcw      tty16   Aug 29 15:06
shaw     tty17   Aug 30 09:02
dell     tty19   Aug 28 16:04
jt       tty1d   Aug 30 09:19
karla    tty1e   Aug 30 10:39
sam      ttyp0   Aug 30 12:50   (triton)
jd       ttyp1   Aug 29 10:12   (venus)
mars%
```

`who` shows you the login-name of each user on the system, the terminal that person is using, when they logged in, and, if logged in from a remote machine, the name of that machine.[33]

From time to time, you may want to see what others are doing. The `w` command tells you what command is running on each user's terminal. In addition, it shows you the amount of time since the user last typed something in (idle), the total CPU time spent by each user so far (JCPU), the CPU time spent by the command now running (PCPU).

[32] If your system uses the yellow pages network services, not all users with access to your system may be listed in this file. To find out more about the yellow pages and users with access over the network, refer to *Using the Network: Beginner's Guide*, or *System Administration for the Sun Workstation*.

[33] See, *Using the Network: Beginner's Guide* for more information about using remote machines.

```
mars% w
  1:18pm  up 4 days,  2:51,  14 users,  load average: 0.32,
0.20, 0.00
User     tty       login@  idle   JCPU   PCPU  what
qv       tty05     8:48am    12     54     14  -csh
patti    tty07     7:57am         6:20     26  mail lisa@sunmark
alder    tty08     9:08am         1:57      8  -csh
domke    tty09     8:44am  3:10     22      4  mail
bartlett tty0c    11:35am  1:40     18      4  -csh
shanda   tty0d     1:02pm    13      7      4  -csh
jd       tty10    11:08am 95:31   2:38   1:18  /usr/ucb/more
gabe     tty13     5:38am     6   2:48     11  -csh
jcw      tty16     3:06pm  2:04   2:38     15  mail
shaw     tty17     9:02am  2:55     24     14  vi eco
dell     tty19     4:04pm 28:59     18      4  -csh
jt       tty1d     9:19am     1     48      8  -csh
sam      ttyp0    12:50pm     1     27      6  w
jd       ttyp1    10:12am  1:51   5:36   1:07  mail
mars%
```

To get a detailed list of everyone's processes, use the command

```
ps -au
```

```
mars% ps -au
USER       PID %CPU %MEM   SZ  RSS TT STAT   TIME COMMAND
sam      19755 49.8 10.0  212  140 p0 R      0:03 ps -au
patti    19751 42.4 15.8  366  226 07 S      0:12 vi mail.record
root     19754  4.8  8.3  232  114 08 S      0:02 /usr/lib/sendmail -bm c2
jd       18732  0.0  0.0  186    0 p1 IW     0:44 mail
alder    19752  0.0  2.2   70   24 08 S      0:00 pmsg
shaw     18085  0.0  0.0  300   86 17 IW     0:10 vi eco
jd        1364  0.0  0.0   86    0 10 IW     0:00 /usr/ucb/more
domke    18516  0.0  0.0  180    0 09 IW     0:00 mail
root     19616  0.0  0.4    0    0 p1 Z      0:00 <exiting>
jd         356  0.0  0.0  184    0 10 IW     1:13 mail
sam      19626  0.0  2.7  178   30 p0 S      0:03 -csh (csh)
alder    19753  0.0  1.6   66   16 08 I      0:00 sh -c /usr/lib/sendmail -bm c2
jd       14061  0.0  0.0  178   12 p1 IW     0:03 -csh (csh)
jcw      16334  0.0  0.0  180    0 16 TW     0:00 mail
jd        1360  0.0  0.0  166   12 10 IW     0:00 sh -c /usr/ucb/more
mars%
```

The -a option tells ps to show you information about all processes, not just your own. The -u option gives a more detailed display that includes the name of the user who owns the process. The -au option is simply the combination of these two.[34] For information about the remaining columns, refer to ps in the

[34] Single-letter options that can be combined like this are sometimes referred to as *flags*.

Commands Reference Manual.

Changing Identity with su

If you know someone else's password, you can temporarily assume that person's system identity by using the su (*superuser*) command. A common reason for doing so is to get access to files that you don't own. Suppose that a colleague has moved a file into one of your directories that you want to edit:

It is usually better to copy such a file yourself, since you often don't know the password of another user.

```
mars% ls -l
total 34
-r--r--r--  1 sam      1697 Aug  2 13:35 env.b
-r--r--r--  1 sam      1244 Aug  2 13:50 chapter.1
-r--r--r--  1 jd       3623 Aug  2 13:50 program.source
```

First, use cp to make a copy of the file. You will own the copy, and can edit it. To get rid of the version you don't own, switch your userid and delete it:

```
mars% cp program.source my.source
mars% su jd
Password: ...
mars% rm program.source
mars%
```

To revert to your previous ID, enter a [CTRL-D] (or the command logout).

If, after switching userids, you want to find you who you are logged in as, type in whoami.

```
mars% whoami
jd
mars% ^D
mars% whoami
sam
```

Or, try the command lines:

```
who am i
```

or

```
who mom likes
```

Becoming root, the superuser

If you omit the *name* argument, su attempts to switch you to root, also referred to as the *superuser*. When you become the superuser, the last character of the prompt changes from a percent sign (%) to a pound sign (#).

```
mars% su
Password: ...
mars#
...
^D
mars%
```

As `root`, you can kill any process running on your machine. You have read and write privileges on every file on your machine's disk (or disk partition) and you can change the ownership of these files.[35]

You must become `root` to perform system maintenance tasks such as adding new users, adding new terminals or printers, etc. Refer to the *System Administration for the Sun Workstation* for more information on performing these tasks.

[35] Files mounted from a remote host belong to that machine. You must be logged in as `root` on the remote host to get superuser privileges for files that reside on it. Refer to *Using the Network: Beginner's Guide* to find out more about remote hosts and mounted file systems.

6

Managing Your Files

6

Managing Your Files

UNIX has good facilities to help you locate files, monitor changes to important files, and manage your space on the disk.

6.1. Locating Files

To locate a file in the file system hierarchy, you may need to know its absolute pathname. When trying to locate a file, chances are that you are either looking for the pathname of a particular command, or you are looking for a certain text file. UNIX provides several ways to locate commands. These are presented first, followed by methods for locating text files.

Looking Up a Command with `whereis` and `which`

To find the pathname of a standard UNIX command, type in `whereis` followed by the command name. (`whereis` also displays the pathname of the `man` entry.)

```
mars% whereis csh
csh: /bin/csh /usr/man/man1/csh.1
```

You can also use `which` to look up a command. This is useful when you have commands that are aliased, or if your system contains commands in addition to the standard set. If the command is an alias, `which` shows you its definition. If the command is in a directory listed in your `path` variable, `which` displays its pathname. If there is more than one version of a command in those directories, `which` displays the version that the system finds first. This is the same version that the system performs when you type the command in.

```
mars% which ls
ls:      aliased to ls -F

mars% which chesstool
/usr/games/chesstool
```

Looking Up a Command's Description with `whatis`

`whatis`, followed by the name of a command, will give you a brief description what that command does.

```
mars% whatis whatis
whatis (1)           - describe what a command is
```

Looking Up Files with `find`

Starting with a named directory,[36] `find` searches for files that meet conditions you specify. A condition could be that the filename match a certain pattern, that the file is owned by a certain user (or belong to a certain group), or that the file has been modified within a certain timeframe.

Unlike most UNIX commands, `find` options are several characters long, and the name of the starting directory must precede them on the command line.

`find` *directory options*

Each option describes a criterion for selecting a file. A file must meet all criteria to be selected. So, the more options you apply, the narrower the field becomes. The `–print` indicates that you want the results to be displayed. (As later on, you can use `find` to run commands. You may want `find` to omit the display of selected files in that case.)

The `–name` *filename* option tells find to select files that match *filename*. To see which files within the current directory and its subdirectories end in `s`, type in:

```
mars% find . -name '*s' -print
./programs
./programs/graphics
./programs/graphics/gks
./src/gks
...
mars%
```

Other options include:

`–name` *filename*	select files whose rightmost component matches *filename*. Surround *filename* with quotes if it includes filename substitution patterns.
`–user` *userid*	select files owned by *userid*. *userid* can be either a login name or user ID number.
`–group` *group*	select files belonging to *group*.
`–mtime` *n*	select files that have been modified within *n* days.
`–newer` *checkfile*	select files modified more recently than *checkfile*.

You can combine options within (escaped) parentheses (`\(...\)`) to specify an order of precedence for criteria. Within escaped parentheses, you can use the `–o` flag between options to indicate that `find` should select files that qualify under either category, rather than just those files that qualify under both.

```
mars% find . \( -name AAA -o -name BBB \) -print
./AAA
./BBB
```

[36] You must supply a name.

You can invert the sense of an option by prepending an escaped exclamation point. `find` then selects files for which the option does *not* apply.

```
mars% find . \!-name BBB -print
./AAA
```

Running Commands with `find`

You can also use `find` to apply commands to the files it selects with the

```
-exec command '{}' \;
```

option. This option is terminated with an escaped semicolon (`\;`). The quoted braces are replaced with the filenames that `find` selects.

You can use `find` to automatically remove temporary work files. If you name your temporary files consistently, you can use `find` to seek them out and destroy them wherever they lurk:[37]

```
find . \(-name test -o -name dummy \) -exec rm '{}' \;
```

Looking at File Types with `file`

Sometimes you want to see what sort of data a file contains without having to look at its contents. In particular, if the file is a compiled program (*object-file*), trying to display its contents can produce spectacular and disconcerting results on your screen. `file` quickly tells you whether a file contains plain text, `troff` sources, C program sources, executable files, or tape-format archives.

```
mars% file *
AAA:  empty
document:  nroff, troff, or eqn input test
troff.output:  troff (CAT) output
program:   demand paged pure executable
scratch:  ascii text
```

6.2. Looking at Differences Between Files with `diff`

It often happens that different people with access to a file make copies of it and then edit their copies. `diff` will show you the specific differences between versions of a file and provide you with an indication of how the contents of one can be edited to produce the other. The command

```
diff leftfile rightfile
```

scans each line in *leftfile* and *rightfile* looking for differences. When it finds a line (or lines) that differ, it determines whether the difference is the result of an addition, a deletion, or a change to the line, and how many lines are affected. It tells you the respective line number(s) in each file, followed by the relevant text from each.

[37] For good housekeeping, you may want to get rid of such files on a regular basis without having to think about it. If you put a command like this in your `.logout` file, then whenever you log out, the system will clean up unwanted files for you.

If the difference is the result of an addition diff displays a line of the form·

l[,*l*] **a** *r*[,*r*]

where *l* is a line number in *leftfile* and *r* is a line number in *rightfile.* If the difference is the result of a deletion, diff uses a d in place of a; if it is the result of a change on the line, diff uses a c.

The relevant lines from both files immediately follow. Text from *leftfile* is preceded by a *left* angle-bracket (<). Text from *rightfile* is preceeded by a *right* angle-bracket (>). This example shows two sample files, followed by their diff output.

Sample 1:

```
mars% cat sched.7.15
Week of 7/15

Day:    Time:   Action Item:        Details:

T       10:00   Hardware mtg.       every other week
W       1:30        Software mtg.
T       3:00    Docs. mtg.
F       1:00        Interview
```

Sample 2:

```
mars% cat sched.7.22
Week of 7/22

Day:    Time:   Action Item:        Details:

M       8:30        Staff mtg.      all day
T       10:00   Hardware mtg.       every other week
W       1:30        Software mtg.
T       3:00    Docs. mtg.
```

diff *output:*

```
mars% diff sched.7.15 sched.7.22
1c1
< Week of 7/15
---
> Week of 7/22
4a5
> M             8:30            Staff mtg.              all day
8d8
< F             1:00            Interview
```

Figure 6-1 *Two Sample Files and* diff *Output*

6.3. Monitor Changes with `sccs`

When you want to protect a file from accidental deletion, keep track of changes to it, or allow more than one person to modify it, you can monitor the file using `sccs`. `sccs`, or "source code control system" is a utility program that protects important files by allowing only one person at a time to make changes, by maintaining a record of those changes, and by rebuilding the current (or any previous) version upon request.

Putting a File Under `sccs` Control (`sccs create`)

To put a file under `sccs` control, perform the following steps:

1. `cd` to the directory containing the file(s) to be protected. If a subdirectory name `SCCS` is not already present, create it. If you want to allow other users access to the files, change the permissions of the current directory and those of the `SCCS` subdirectory to `775`.[38]

```
mars% cd project
mars% mkdir SCCS
mars% chmod 775 . SCCS
```

2. Type in a command of the form:

   `sccs create` *filename*...

   filename is the name of a file or files to monitor.

```
mars% sccs create *
```

   For each file that you indicate on the command line, `sccs` produces a special file called a *history* file, and puts it in the `SCCS` subdirectory. The history file has a name of the form:

   `s.`*filename*[39]

   and contains a complete record of all lines changed throughout the life of the file. `sccs` maintains a checksum on all history files, so *do not* edit them! `sccs` may respond with the warning:

   `No id keywords (cm7)`

   This message can safely be ignored when you are auditing your own files.

When working with files that are part of a large project, `sccs` ID keywords can be important. Refer to *Programming Utilities for the Sun Workstation* for more information about `sccs` as a tool for managing large programming projects.

3. Remove the backup file(s) that `sccs` leaves behind. These files are created by `sccs` as a safety precaution, and are no longer necessary once the `create` operation is complete. Names of these backup files begin with a comma (`,`).

```
mars% rm ,*
```

[38] Unless you are sure that you do *not* want them to have access, it is normally a good idea to change permissions of both directories to allow it, at least for other members of your user group.

[39] History files are also referred to as "s.files."

Once under `sccs` control, you have to check a file out before you can make changes to it. Files that aren't checked out through `sccs` have permissions set to read-only for everyone (`444`).

Which Files are Checked Out? (`sccs info`)

To see which files in the working directory are checked out, use the `sccs info` command. If no files are checked out, `sccs` responds with the message:

```
Nothing being edited
```

If there are files checked out, it lists those that are, the current version number of each, the version number each will have when checked in again, the name of the user who checked out each and the date and time of check-out:

```
csh.1: being edited: 1.4 1.5 sam 85/09/04 16:32:15
```

Recovering the Current Version (`sccs get`)

Because several people may have write access to the directory, it is possible that a file in the working directory may be deleted accidentally. Files that *aren't* under `sccs` control are gone for good once they are removed, by you can easily restore files under `sccs` from their history-files using the `sccs get` command:

`sccs get` *filename*

If you want to recover the current version of all files in the directory, use the command:

```
sccs get SCCS
```

Checking a File Out (`sccs edit`)

Only one person at a time can check a file out. This assures you that changes won't be lost, garbled, or intermixed between the edits of different users. To check out a file, type in `sccs edit` followed by the file or files you wish to check out. `sccs` will respond with the current version number, the new version (delta) number, and the number of lines in the file.

```
mars% sccs edit program
1.1
new delta 1.2
220 lines
mars%
```

Once checked out, you can edit the file using `vi`, or an editor of your choice.

When you check out a file, `sccs` changes the ownership of the file to you, gives you write permission (owner only), and places a *lock* file containing your userid, the version number, and other information in the `SCCS` directory.[40] When you check the file back in, the lock file is removed and the permissions are set to read only, but you retain ownership of the file.

[40] The lock file has a name of the form: `p.`*filename*, and referred to as a "p-file."

Looking at Current Changes (`sccs diffs`)

While still checked out, you may want to review the changes you have made so far. To do so, type in:

```
sccs diffs filename
```

`sccs` responds with standard `diff` output.

Checking a File In (`sccs delget`)

When you are done making changes you can check in the new version of the file by typing in the nonintuitive command:

```
sccs delget filename
```

`delget` is a contraction for `delta`, the command to incorporate a new version into the history file, and `get`, the command to recover the newest version (that you are just now checking in).[41]

When you use `delget` (or `delta`) to check in the file, `sccs` asks you for a line of comments. These comments are included in the history file, and should briefly summarize the changes you have made. After adding your comments and pressing RETURN, `sccs` responds with the new version number, the number of lines inserted, deleted and unchanged, and the total number of lines.

```
mars% sccs delget program
comments? added remarks for more readable code
1.2
43 inserted
18 deleted
287 unchanged
1.2
348 lines
```

Backing Out With No Changes (`sccs unedit`)

To check a file back in without any changes, type in:

```
sccs unedit filename
```

Looking at the File's History (`sccs prt`)

To review a file's history, use the command:

```
sccs prt filename
```

This command shows you the version number, comment lines, date checked in, and user responsible for each version of the file.

[41] If `sccs` responds with an error message, it does not perform the `get` action, and you may have to recover files using `sccs get SCCS`.

```
mars% sccs prt program
SCCS/s.program:

D 1.2 85/09/04 12:51:07 sam 2 1 00042/00008/00357
MRs:
COMMENTS:
added remarks for more readable code

D 1.1 85/08/30 16:54:57 sam 1 0 00365/00000/00000
MRs:
COMMENTS:
date and time created 85/08/30 16:54:57 by sam
```

Comparing Versions (`sccs sccsdiff`)

To compare previous versions of a file, use the command

`sccs sccsdiff -r`*`x.y`* `-r`*`m.n`* *filename*

Where *x.y* and *m.n* are version numbers to be compared. This command produces standard `diff` output.

Restoring a Previous Version (`sccs get -r`)

If you want to back out a version of the file that is already checked in, you must perform the following steps:

1. Recover the previous version. You can look up its number using `sccs prt` *filename*. To rebuild the previous version, type in a command of the form:

   `sccs get -r`*x.y* *filename*

   where *x.y* is the desired version number.
2. Rename the recovered version of the file

   `mv` *filename* `temp`
3. Check the file out with `sccs edit`.
4. Replace the checked-out version with the old version:

   `mv temp` *filename*
5. Check the file back in with `sccs delget`.

To assure that it all worked properly, compare the latest version with the desired previous version using `sccs sccsdiff`.

The typical flow of events when making changes to a file under sccs control is:

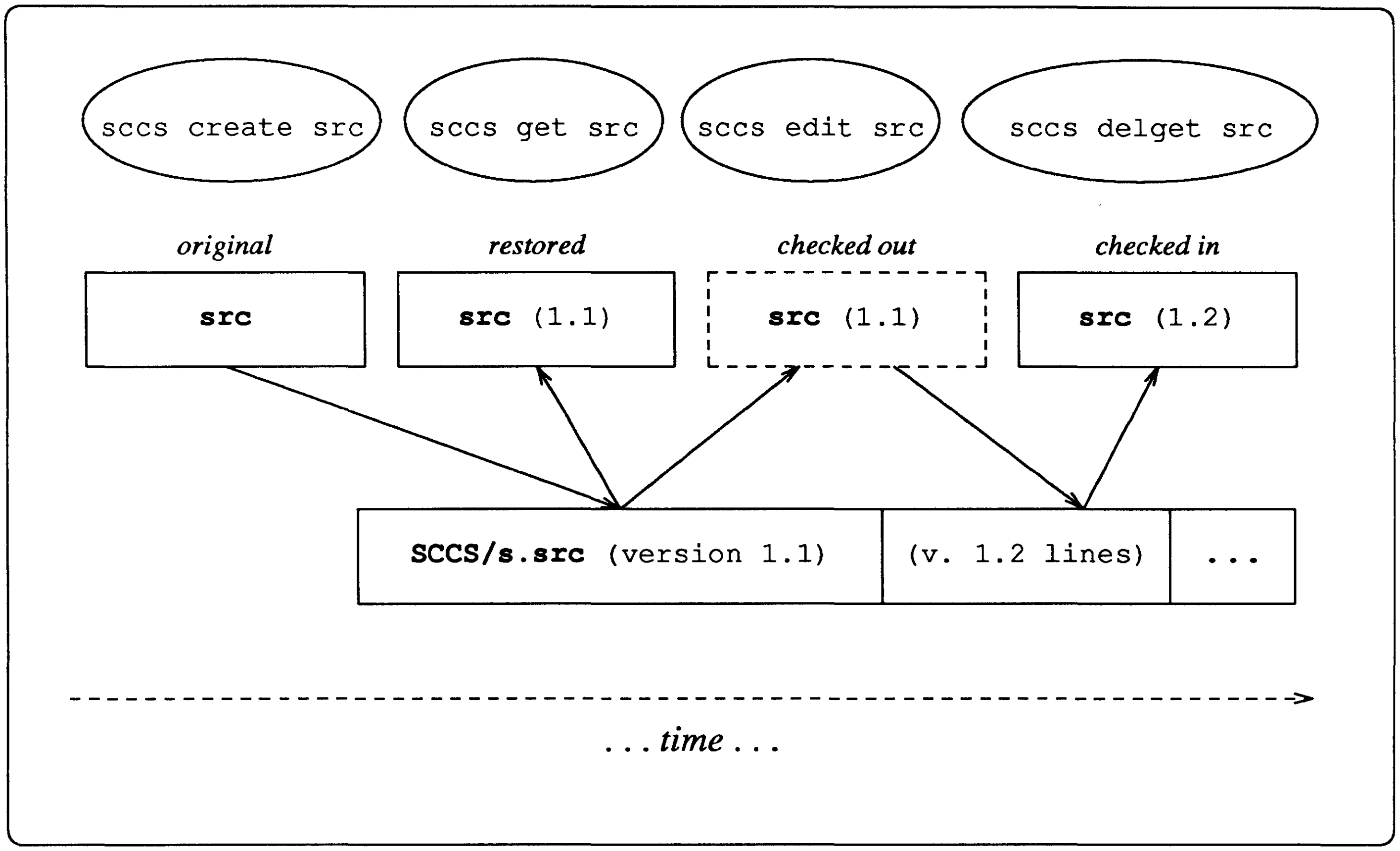


Figure 6-2 *Flow of Events with* sccs *Controlled Files*

Solving Problems with sccs

sccs is a complicated and verbose utility. There may be times when it responds with an error message even though things worked properly. Its error messages are sometimes difficult to interpret. If you are not sure that sccs succeeded in doing what you asked, you can take certain steps to verify whether it has:

Are Files Under sccs Control?

`ls -l SCCS`
will show an s.file for each file under sccs control.

Is the File Checked Out?

`sccs info`
will show which files are checked out, and to whom.

Was the File Checked In?

`sccs prt` *filename*
will show your comments in the first three lines when you have checked in a file successfully.

What If I Can't Check the File Out?

If you attempt to check a file out and you get the message:

```
ERROR [SCCS/s.filename]: writable 'filename' exists (ge4)
```

this usually means that someone has the file checked out already. You can verify this using `sccs info`. If `sccs info` does not list the file as being edited, then the lock file in the SCCS directory has been deleted. When this happens `sccs` will not allow anyone to check the file either in or out.

To correct this problem, first run `sccs diffs` on the file to see if it differs from the version last checked in. If so, it is a good idea to contact the file's owner to find out if the changes made should be kept. If so, then copy the file to a new filename, remove the writable original, and check the file out using `sccs edit`. Then move the new filename back to the original name (overwriting the checked out version), and check the new version back in using `sccs delget`.

If the changes need not be saved, you can correct the problem by simply removing the writable file, restoring the current version using `sccs get` and then checking it out using `sccs edit`.

6.4. Automating Complicated Tasks with `make`

Performing complicated tasks, such as producing object code for programs or formatting large documents involves processing different files through various programs at the proper times and in the proper order. This can be a lot to remember. `make` simplifies these complications by following a a record of the steps involved, called a *makefile*, that you create.

The makefile contains a list of the steps called *targets*. Each target contains a list of UNIX commands; this list of commands is called a *rule*. A target can be qualified by a list of other targets upon which it depends. One target is said to depend on another if the latter must be be completed before the former can be performed successfully. The latter target is called a *dependency*.

For example, an SCCS subdirectory must be created before you can put files under `sccs`. And, you must put a file under `sccs` with `sccs create` before you can check that file out. So the command `sccs edit` depends in practice on the commands `mkdir SCCS` and `sccs create` for its own success.

`make` uses the list of targets as a recipe to produce a desired program, document, or other object file called a *target file,* or simply *target.*

`make` performs only those steps that are required to bring the target files up to date. It lists the various steps involved, and how they depend on one another, and then examines the list to see which target files are outdated.

A target is considered to be outdated when a source file used to produce it has changed since the target file itself was last produced. `make` then performs only those steps required to replace any outdated target files.

`make` has a facility to perform *macro substitution*.[42] This allows you to abbreviate long lists, and to predefine parameters that often change, so that with a few simple edits the same procedure can be used to produce other, similar objects.

[42] Like an alias, a *macro* is a string of text that is replaced by its definition, or *expansion* when encountered in an input file (or command line).

Makefiles

Like a recipe card, a makefile is composed of two sections. The first section is a list of macro definitions. These are described in detail later on. The second section outlines steps in the procedure and their relationships to one another. In `make` parlance, each step is called a *target.*

Each target has a name. If that target's function is to produce an object file of some sort, then the name of the target should be the same as the name of the file it produces. If the target performs some sort of housekeeping step, then it can have any name you like.

A target may also have a list of *dependencies,* or targets it depends on, associated with it. `make` uses this list to determine whether files produced by the target are up to date.

Finally, each target has a list of UNIX commands to perform. When performing a step, `make` performs each command in turn, starting a Bourne Shell[43] for each command line.[44]

The following is an example of a makefile to put the contents of a directory under `sccs` control. The file consists of just three targets, and no macro definitions:

```
# makefile: for putting files under sccs

#         no macro definitions

#         target definitions

put.under:  SCCS
    # these lines begin with a required tab character
    -sccs create *
    -rm ,*
    -sccs get SCCS

SCCS:
    -mkdir SCCS
    -chmod 775 SCCS
```

Figure 6-3 *Sample Makefile to Put Files Under* `sccs`

The targets are `put.under` and `SCCS`. The target `put.under` depends on the target `SCCS`. If the `SCCS` directory is not already present and up to date (directories always are), `make` performs the commands listed under `SCCS` first.

The format of each target is significant. The name of the target must be followed by a colon and the list of dependencies, if any. (If this list is longer than one line,

[43] Because it runs a Bourne shell, certain C-Shell constructs, such as `foreach`, don't work. Refer to `sh` in the *Commands Reference Manual* for more information about the Bourne Shell.

[44] Since each command line is executed in its own Shell, you must use the command separation character ;, and the command-line continuation character \ RETURN to build command *routines*.

then escape the carriage return with a backslash.) The list of commands immediately follows the target name, and each command line begins with a TAB.

Comments begin with a #, and can be placed to the right of commands on any line (not ending in a backslash). At least one blank line separates target definitions from one another.

When you prepend a - to a command, make ignores a nonzero (error) return code from that command. Normally, make halts whenever a command it runs exits with a nonzero status. Adding the dashes in this case tells make to continue putting new files under sccs control, even though it may encounter older files already there.

Because make checks for dependencies, you can write makefiles in a top-down fashion. The step that produces the final output should appear first. Steps that it depends upon can appear next, followed by steps that *they* depend on.

Running make

When the makefile is ready, simply type in make.

make looks for a file in the working directory named makefile, or Makefile,[45] checks for dependencies, beginning with the first target it encounters, and then performs commands in their proper order.

```
mars% make
mkdir SCCS
chmod 775 SCCS
sccs create *

SCCS:
ERROR: directory 'SCCS' specified as 'i' keyletter value (ad29)

makefile:
No id keywords (cm7)
messages from sccs
rm ,*
sccs get SCCS
messages from sccs
mars%
```

The error message

```
ERROR: directory 'SCCS' specified as 'i' ...
```

indicates that sccs attempted to create a history file for the directory SCCS. Because we used a dash as the first character of the command line, make continued processing.

[45] You can specify the name of some other makefile, using the -f *filename* option: make -f buildit

Testing Makefiles

Most makefiles take a bit of debugging. To find out what commands make will perform without actually running them, use the –n option.

```
mars% make -n
sccs create *
rm ,*
sccs get *
```

In the above makefile, put.under depends upon SCCS. When you ran make the first time, the SCCS directory was created. When you ran make -n subsequently, make did not indicate that it would perform that step (since it was up-to-date anyway). If you were to remove the SCCS directory, and then run make, it would perform commands in the SCCS target once again.

Defining Macros in the Makefile

The next example is a makefile used to format and print a document made up of several source files. With macro substitution, copies of a makefile such as this can be used for different documents:

```
# Makefile: for printing a document

#         macro definitions

SOURCES = title intro tutorial reference appendix
PRINTER = Plw
MACROS  = ms

#         target definitions

print: troff.output
        lpr -$(PRINTER) -t troff.output &

troff.output: $(SOURCES)
        tbl $(SOURCES) | eqn | troff -t -$(MACROS) > troff.output
```

Figure 6-4 *Sample Makefile for Printing a Document*

A change to the list of sources, the printer, or the macro package can be made in one place and take effect throughout the makefile. For large and complex procedures, this is a big advantage.

By placing the troff output in an intermediate file,[46] you can avoid having to reformat the document every time you want to print a copy. By making print depend upon the file troff.output, you can be sure that you always get the latest formatted version.

[46] troff intermediate output files are *not* text files. They will produce strange results if you try to look at them on the screen, and they should *not* be placed under sccs. It would be a good idea to put the source files under sccs instead.

By making `troff.output` depend on the list of sources (the expansion of the `$(SOURCES)` macro), you can be sure that when you change any one of the sources, the change will be reflected when you print the document.

Selecting A Target

You can select any target in the makefile by specifying it as an argument to `make` on the command line. If a target does not appear in the list of dependencies for the target you select (or the first target by default) `make` will not perform it. So, you can record several independent procedures within the same makefile. For example, this makefile can be used either to put new source files under `sccs`, or to print a finished document.

```
# Makefile: for printing a document
#           and putting sources under SCCS

#         macro definitions

SOURCES = title intro tutorial reference appendix
PRINTER = Plw
MACROS  = ms

#         target definitions

print: troff.output
        lpr -$(PRINTER) -t troff.output &

troff.output: $(SOURCES)
        tbl $(SOURCES) | eqn | troff -t -$(MACROS) > troff.output

# -------------------------------------------------------

put.under: SCCS
# the next three lines begin with a tab
        -sccs create `ls | grep -v troff.output`
        -rm ,*
        -sccs get *

SCCS:
        mkdir SCCS
        chmod 775 SCCS
```

Using this makefile, if you type in `make`, you will get the document. If you type in

```
make put.under
```

your sources will be put under `sccs`.

6.5. Managing Disk Storage

Space on the disk is a limited resource. So, it is a good idea to keep track of how much space you use, especially if your system is running with disk quotas.[47]

UNIX provides facilities to monitor your disk usage and locate big directories that are candidates for housekeeping. Even so, it can be unwise to delete old files willy-nilly. You never know what gems you may have socked away there. So, the system also provides a facility to make tape archives of important files. Tape archives are especially good for large files that you need to keep but don't often use. If you make a tape archive before cleaning house, you can be sure that you won't lose anything important. You can use `df`, `du` and `ls -l` to locate such files, and you can use `tar` to move them onto a tape for storage offline, as described in the following sections.

Looking at Disk Usage with `df`

`df` shows you the amount of space used up on each disk that is *mounted* (directly accessible) to your system. It is very simple to use, just type

```
df
```

to see the capacity of each disk mounted on your system, the amount available, and the percentage of space already used up.

```
mars% df
Filesystem           kbytes    used   avail  capacity  Mounted on
/dev/nd0               4771    2197    2096     51%    /
/dev/ndp0              5691    4010    1111     78%    /pub
titan:/usr.MC68010    53007   42871    4835     90%    /usr
topaz:/usr/topaz     318943  236688   50360     82%    /usr/topaz
panic:/usr/games     117259   67484   38049     64%    /usr/games
panic:/usr/man       117259   67484   38049     64%    /usr/man
opium:/usr/opium     327599  214546   80293     73%    /usr/opium
athena:/usr/doc      105843   59006   36252     62%    /usr/doc
athena:/usr/athena   266107  219747   19749     92%    /usr/athena
titan:/usr/doctools   15887   11604    2694     81%    /usr/doctools
```

Filesystems at or above 90% of capacity should be cleansed of unnecessary files. You can do this either by moving them to a disk that is less full using `cp` and `rm`. You can make a tape archive and then remove them. Or, you can simply remove them outright. Of course, you should only perform housekeeping chores on files that you own.

Directory Usage and `du`

You can use `du` to display the usage of a directory and all its subdirectories (in kilobytes).

`du` shows you the disk usage in each subdirectory. To get a list of subdirectories in a filesystem (disk), `cd` to the pathname associated with that filesystem, and run the following pipeline:

[47] A disk quota is a limit on the amount of space (information) a user is allowed to use on the disk at any one time.

```
du | sort -r -n
```

For instance:

```
mars% du | sort -r -n
5314    .
1155    ./Documents.new
818     ./SCCS
234     ./Programs.new
230     ./Reference.new
204     ./Reference.old
123     ./Library.new
89      ./Library.old
87      ./Users.Guide.old
49      ./Reports.old
27      ./Documents.old
5       ./Programs.old
```

This pipeline, which uses the *reverse* and *numeric* options of `sort`, pinpoints large directories. Use `ls -l` to look at the size (in bytes), and modification times of files within each directory. Old files, or text files over 100K bytes, often warrant storage *off-line*.

6.6. Making a Tape Archive with `tar`

The simplest and most complete method to make a tape archive is to:

1. Mount a fresh tape on the tape drive. If you don't know how to do this, see your System Administrator or consult *System Administration for the Sun Workstation* for details.
2. `cd` to a directory you wish to archive. If you wish to archive an entire hierarchy of files, `cd` to the topmost directory in that hierarchy. `tar` will archive the directory and all its subdirectories.
3. Type in the `tar` command as follows:

   ```
   tar -cvf drive
   ```

   The `-c` option tells `tar` to *create* a new tape archive and overwrite the previous contents of the tape. The `v` stands for *verbose*. `tar` tells you everything that it is doing. The `f` tells `tar` to put the archive on the file (tape drive) *drive*. Your System Administrator can tell you the name of a tape drive to use.

Tapes can be reused. If you do not wish to overwrite the previous contents, you can use `-r` rather than `-c`. With `-r`, `tar` skips to the end of the previous archive, and then adds files onto the end. If you want to conserve space on the tape, you can use `-u`.[48] With `-u`, `tar` replaces files whose contents have changed with their newest version, adds new files onto the end, and leaves untouched files alone.

[48] The `-r` and `-u` options do not work with quarter-inch cassettes. They only work with half-inch tape drives.

drive can be a diskfile. Since `tar` output takes up less space than do text files, a tape archive on disk can provide some space savings and a bit more convenience than using an actual tape. For even more space reduction, run the tape archive file, or *tarfile* through `compact`.[49]

Looking at the Contents of a Tape Archive

To examine the contents of a `tar` tape archive, use the `–t` option:

```
tar -tvf drive
```

To search for a specific file on the tape, pipe the output of `tar -t` through `grep`.

Extracting Files From a Tape Archive

To extract files from a tape archive, `cd` to the directory in which to place the file, mount the tape, and then use the `tar -x` option:

```
tar -xvf drive filename ...
```

If you omit filename, `tar` extracts the contents of the entire tape. If you specify a *filename*, or a list of filenames, `tar` extracts the named file(s).

[49] The command `uncompact` restores the tarfile to its original state, and you can then use `tar` to retrieve files from within the tarfile just like you would from a tape drive.

7

More About Printing

7 More About Printing

In *Getting Started With UNIX* you learned how to print a file. Printers are often in high demand, and are normally shared by a number of people. To keep things running smoothly, the system feeds each request to the printer on a first-come first-served basis. Requests that are waiting are kept in the print queue.

7.1. Looking at the Queue with `lpq`

To look at the queue on the printer you normally use, type in

```
lpq
```

(short for "line printer queue"). If the queue is empty, `lpq` will respond with:

```
no entries
```

If there are some entries, `lpq` will list them for you and indicate which one is currently being printed.

```
mars% lpq
Rank   Owner      Job  Files                   Total Size
active sam        18   standard input           39668 bytes
1st    sam        19   document                443820 bytes
2nd    joe        20   program.listing          32833 bytes
...
```

7.2. Removing Printer Jobs with `lprm`

If you decide not to print a job after all, you can remove it from the queue by typing in `lprm` followed by the job number:

```
mars% lprm 19
dfA019mars dequeued
cfA019mars dequeued
mars% lpq
Rank   Owner      Job  Files                   Total Size
active sam        18   standard input          39668 bytes
1st    joe        20   program.listing         32833 bytes
```

To remove all your jobs from the queue, use the – option:

```
lprm –
```

7.3. Selecting a Printer `lpr -P`

If the line for the printer is too long and there is another printer available to your system, you can direct jobs to that other printer with the –P*printer* option of `lpr`. Your System Administrator can tell you the names of other printers that you can use. `lpq` and `lprm` also accept this argument.

```
mars% lpr -Plaserwriter memo
mars%  lpq -Plaserwriter
Rank   Owner      Job  Files                   Total Size
active jd          98  standard input       559668 bytes
active jenny       99  memo                   2077 bytes
active louisf     100  letter                57320 bytes
active sam        115  document             621633 bytes
mars% lprm -Plaserwriter 115
lrhost: dfA115mars dequeued
lrhost: cfA115mars dequeued
```

7.4. Printing `troff` Output Files with `lpr -t`

To print `troff` output files, use the `–t` option of `lpr`.

```
lpr -t troff.output
```

7.5. Printing Screen Dumps

If you want to capture an image of the workstation screen on paper, use the following pipeline:

```
screendump | rastrepl | lpr -v &
```

`screendump` captures the image dot-for-dot, `rastrepl` increases its size, and the `-v` option of `lpr` prints the resulting image. There is significant computation involved in each of these steps, so be sure to run this pipeline in the background.

7.6. Printing Other Graphics Displays

`lpr` will print out a variety of graphics displays, depending upon the capabilities of the printer you use. For more information, consult the *Commands Reference Manual*, and your System Administrator.

A

Glossary

A

Glossary

angle-brackets
Term for the characters < and >.

append
To add text or data onto the end of a file.

archive
A copy of a file or set of files, usually on tape, made for historical purposes or for long-term storage.

background
A process that is running, but does not have control of the terminal from which it was started, is said to be running in the background.

braces
Term for the characters `{` and `}`.

brackets
Term for the characters `[` and `]`.

builtin
Adjective for a command that is part of a particular shell; it is literally "built in" to the shell software. Such commands are only available when using the particular shell that supports them. Contrast this with such commands as `ls`, which is available for use with either shell.

C-Shell
A command interpreter for UNIX that provides filename substitution, alias substitution, a history mechanism, variable substitution, command (output) substitution, and job control. The C-Shell can interpret commands directly from the terminal, or from command files with a syntax modeled after the C programming language.

child process
A process started from within shell or other process.

contents
The text or data contained in a file.

default
An assumed value, or an action taken when you omit an argument, command, or value.

dependency
A step within a procedure upon which a subsequent step depends. The step must be completed before the latter can be performed properly. `make` uses this notion to organize sets of UNIX commands, and do the minimum amount of work required to perform a task or bring a set of object-files up to date.

device
Typically a hardware peripheral supported by the system, and the software that controls it. May also be a specialized software program. UNIX treats a device as if it were a file. The programs that operate peripheral devices reside in the directory `/dev`.

directory
A type of file that contains names and access information about other files, including other directories. Directories are organized in a hierarchy, the root of which is named `/`.

drive
(*tape drive* or *disk drive*). The hardware that performs the physical transfer of data from the system onto a tape or disk, and vice-versa.

embedded
Contained within a file, within a line of text, or within a word. Usually applied to commands or symbols that are surrounded by ordinary characters.

encrypt
To encode or scramble data to prevent unauthorized reading.

environment
General: to the extent that an interactive program can be customized, the values of the various options, settings, and variables that are currently in effect. Technical: the set of data inherited from the parent process and/or passed along to child processes.

escape
A character, usually a backslash, indicating that the character following it is to be interpreted as plain text, rather than as a symbole having special meaning.

event
In history substitution: the text of a command-line contained in the history list.

execute
To perform a set of instructions or program.

expansion
The value of a variable or macro. For instance, in the C-Shell the expansion of the character ~ is the pathname of the user's home directory.

filename
The name of a file, directory, or device.

file
A portion of a mass-storage memory device, typically a disk, containing a specific, named set of data. Generalized to include any source from which data can be received or transferred within the system.

file type
A field in the permissions column of the `ls -l` display that indicates whether the file is a plain disk file, a directory, a device, or a symbolic link.

filter
A command or program that accepts text from the standard input, applies a transformation rule (or rules) to that text, and produces text on the standard output.

foreground
The process that has control of the terminal is said to be running in the foreground. Process that do not control the terminal are said to be running in the background.

fork
By a shell or command: to start a new process and wait for it to finish before proceeding.

group
A subset of users with access to the system. Members of a group may be granted more complete access to files than the public at large. The permissions that control group access to files.

job
A background process, running or stopped, under the control of the C-Shell.

key
A character string used to encode or decode a file by `crypt`.

link
A filename, or entry in a directory corresponding to a file. A *hard* link is a direct entry. A symbolic link is a string that contains the name of the file it is associated with.

macro
A string of text that is replaced by another, typically much longer, string when interpreted by a shell or program.

makefile
A file containing instructions for `make`. Typically named `makefile` or `Makefile`.

modification time
The date and time at which a file was last changed. A field in the directory entry for a file that can be altered directly using the `touch` command.

monitor (v.)
To maintain a record of changes to a file, to assure that only one user at a time can make changes, and to assure that the most recent version of a file can quickly be restored.

multitasking
Performing multiple tasks at once. The ability of the system to handle the work of several simultaneous users or windows.

noninteractive
A program that accepts no input from, and displays no output on, the terminal.

object-file
A file containing the output, typically not text, of a compiler, plotting program, or other such program.

off-line
Disconnected from the system.

operation
The action of the system or program to accept input, transform data, and produce output.

owner
The user to whom a file belongs, who can alter its name, access permissions, and other attributes.

pattern
A string that includes special characters that, when interpreted, correspond to a set of possible text strings.

parent directory
A directory containing the current directory, or directory of interest.

parent process
A process, from which the current process of interest was started.

permissions
Attributes of a file that determine whether a specific user has access to read, write on (or delete), or execute (use as a command), a file.

pipe
The vertical bar character |. The mechanism by which the system passes the output of one command as direct input to another command.

pipeline
A set of commands connected by pipes. The intermediate commands are typically *filters*.

process
General: A command that is being performed by the system. Each process has a unique number. The mechanism by which the system keeps track of a single task among the many requested of it at any given time. Technical: a set of instructions and data under the control of the system's memory management facilities.

public
The entire set of users who have access to the system. The permissions that control public access to files.

range
A set of characters specified by the first and items in a list. For instance, the entire upper-case alphabet can be specified as: `A-Z`.

redirect
The standard input, standard output, and standard error output of a command is normally received by, or sent to, the terminal. To explicitly indicate a file from which, or to which the command is to send or received data using symbols such as > and <.

regular expression
The method for specifying search patterns for `grep`, and editors such as `vi`.

resources
Refers to the computation capacity and speed, available memory, (and sometimes the peripheral devices) available to the system.

return code
The value returned (to its parent) by a process upon completion.

robust
Programs: Able to perform reliably under a variety of conditions, or with a variety of (possibly unexpected) data. Syntax: The degree to which a set of rules allows for expression of a wide range of information.

routine
A set of commands or instructions that together perform a complete task.

s-file
An `sccs` history files in the `SCCS` subdirectory.

shell
A programmable command interpreter.

size
The number of characters in a text file.

standard error
The channel through which a command sends diagnostic messages.

standard input
The channel through which a command receives data.

standard output
The channel through which a command sends results.

state
The current condition of a process.

string
A set of characters terminated on either end by a tab, space, newline, or other delimiting character.

subdirectory
A directory that resides within another. For instance, `/usr` is a subdirectory of `/`.

subshell
A shell invoked from within another shell or program.

superuser
Another name for the `su` command. The ability to temporarily adopt the ID of another user on the system. A term for the Operator or System Administrator's userid, `root`.

rule
A list of UNIX commands for `make` to perform in order to complete a step, or produce a target file.

syntax
General: the format for a legal command and its arguments. Technical: the rules by which input is interpreted.

target
An object file to be produced, or label for a list of UNIX commands to be performed, by `make`.

user
A person with an account on the system who can log in, issue commands, and create files.

userid
The login name, or ID number assigned to each user by the system administrator.

variable
A named location in which a data value (or list of values) is temporarily stored in memory.

B

C-Shell Scripts

B

C-Shell Scripts

You can put a sequence of UNIX commands in a file called a *script*. By using the `source` *filename* command, or by setting the execute permissions and typing in the filename as if it were a command, you can tell the C-Shell to read and perform commands in the file.

NOTE *We recommend that you use the Bourne shell for writing shell scripts. The Bourne shell has a simpler command syntax, faster execution time, and provides better security. Refer to Appendix F for information about writing Bourne shell scripts.*

This appendix outlines features that you can use when writing scripts for the C-Shell.

C-Shell Invocation

C-Shell scripts do not serve the same function as `make`, which is useful for consistently performing a set of operations on related files. While scripts can be written to do this, the C-Shell is more general in scope. Scripts do not check for dependencies, for instance. And, there are many things that you can do with scripts, such as prompting for input from the terminal, that are not practical using `make`.

When a script is invoked by name, the C-Shell looks at the very first line of the file to decide how to run it:

- ☐ If the first line starts with a # (hash sign), the system uses the C-Shell to run the script.
- ☐ If the first line does *not* start with a # (hash sign), the system uses the Bourne shell to run the script.
- ☐ If the first line of the script starts with a `#!`, followed by the name of a program, the system uses that program to perform commands in the script.

To run a script with no C-Shell startup processing, the first line should be of the form:

```
#! csh -f script
```

Command-Line Arguments in Scripts

To pass command-line arguments as parameters to a script, type its name, followed by any arguments you wish. The C-Shell places words following the name in the variable `argv`, the *arguments list*. Command-line arguments are treated as words contained in this variable, or you can use the equivalent variables: `$1` through `$`*n* where *n* is the number of arguments in the list.

Variables in Scripts

A number of notations are available for accessing words in variables, and other variable attributes. The notation:

```
$?name
```

expands to `1` if a named variable exists (using the `set` command), or to `0` otherwise.

```
mars% set var=(a b c)
mars% echo $?var
1
mars% unset var
mars% echo $?var
0
```

All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the count of words in the variable *name*:

```
mars% set var=(a b c)
mars% echo $#var
3
mars% unset var
mars% echo $#var
var: Undefined variable.
```

To expand to the process number of the C-Shell performing the script, use:

```
$$
```

Since this process number is unique in the system, it can be used to generate unique temporary file names.

The redirection characters:

```
$<
```

indicate that a line is to be read from the terminal. To write out the prompt `yes or no?` without a newline and then read the answer into the variable `a`:

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

In this case `$#a` would be `0` if either a blank line or CTRL-D were typed in response.

A minor difference between `$`*n* and `$argv[`*n*`]` is that `$argv[`*n*`]` yields an error if *n* is larger than the word count `$#argv`, while `$n` never yields a subscript-out-of-range error. This is for compatibility with older shells.

It is never an error to give a subrange of the form *var*`[`*n*–`]` . If there are less than *n* words in the given variable, then no words are selected.

A range of the form *var*`[`*m*–*n*`]` likewise returns a value without an error, even when *m* exceeds the number of words, provided that *n* is in range.

Expressions

All of the arithmetic operations of the C language are available in the C-Shell with the same precedence that they have in C. These operations are useful for evaluating expressions in branches and loops. The operations == and != compare strings, and the operators `&&` and `||` implement the logical *and* and *or* operations, respectively. The operators =~ and !~ are similar to == and !=, allowing for pattern matching as with filename substitution.

File Enquiries

The expression:

`-e` *filename*

returns `1` if the file exists, and `0` otherwise. Similar primitives provide other tests:

`-r` `1` if read-access is allowed for the user running the script.

`-w` `1` if write-access is allowed for the user.

`-x` `1` if execute-access is allowed.

`-o` `1` if the user owns the file.

`-z` `1` if the file has zero length.

`-f` `1` if a plain file.

`-d` `1` if a directory.

Pathname Processing Primitives

There are also primitives to apply to pathnames to strip off unneeded components:

`:t` removes all but the rightmost component of the pathname.

`:r` removes suffixes beginning with a dot (`.`).

`:e` removes prefixes ending with a dot.

`:h` removes the last component, leaving the pathname of the directory in which the file resides.

Return Codes

It is possible to test whether a command terminates normally by using a primitive of the form `{` *command* `}`, which returns `1` if the command exits normally (with exit status `0`), or `0` if the command terminates abnormally (with a nonzero return code).

If more detailed information about the status of a command is required, it can be executed and the variable `status` examined in the next command. Since every command returns a value to `status`, you must save values of interest on the very next line of the script:

```
set checkpoint=$status
```

where *checkpoint* is a suitable variable name.

Sample C-Shell Script

The following script, `copyc`, copies files named as arguments into a backup directory:

```
#
# copyc copies files named on the command line
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

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

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

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

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

Figure B-1 `copyc` — *Sample C-Shell Script*

Basic Control Structures: `if` and `foreach`

This script uses the `foreach` command, which causes the C-Shell to execute the commands between it and the corresponding `end` with the named variable taking on each of the values given between `(` and `)`. The named variable — in this case `i` — is set to successive words in the list. Within this loop you can use the `break` command to stop executing the loop and `continue` to terminate one iteration and begin the next. After the `foreach` loop, the iteration variable (`i` in this case) has the value it had during the last iteration.

The variable `noglob` is set to prevent filename expansion from being performed on members of `argv`. This is a good idea, in general, if the arguments to a C-Shell script are filenames that have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a `$` variable expansion, but this is harder and less reliable.

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

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

The placement of the keywords here is *not* flexible. The word `then` *must* appear on the same line as `if`, when used with a block of commands.

The C-Shell does *not* accept the formats:

```
if ( expression )
then
```

or

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

For individual conditional commands, the C-Shell has another form of the `if` statement:

```
if ( expression ) command
```

which can also be written as

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

The newline is escaped here for the sake of appearance. The command must not involve `|`, `&` or `;` and must not be another control command. The final \ must immediately precede the end-of-line. This is the only form of the `if` command that can be used within an alias definition.

The more general `if` statement also admits a sequence of `else–if` pairs followed by a single `else` and an `endif`.

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

Introducing Comments with #

The character # introduces a C-Shell comment in a script (but not from the terminal), and the C-Shell ignores all subsequent characters the line.

Other C-Shell Control Structures

The C-Shell also has the control structures `while` and `switch` that are similar to those in C.

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

and

```
switch ( word )

case str_1:
     commands
     breaksw

  . . .

case str_n:
     commands
     breaksw

default:
     commands
     breaksw

endsw
```

See the *csh* manual page for details. C programmers should note that `breaksw` exits from a `switch`, while `break` exits a `while` or `foreach` loop.

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

```
loop:
     commands
     goto loop
```

Here Documents

A *here document* is a special notation used to pass instruction along to commands that normally run interactively. The here document begins with a <<*eot* and ends with a line containing *eot* by itself. *eot* can be any string.

Here is a script that runs `ed` to delete leading blanks from every line in each file in the argument list:

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

The notation << `'EOT'` means that the standard input for the `ed` command is the text in the C-Shell script file up to the next line consisting of exactly `'EOT'`. The fact that the `EOT` is enclosed in quote characters prevents the C-Shell from substituting variables on the intervening lines. In general, the C-Shell uses << to terminate the text to be given to the command. If any part of the phrase following the << is quoted, these substitutions are not performed. In this case, since the form `1,$` was used in the editor script, you needed to ensure that the `$` is not variable-substituted. You can also ensure this by preceding the `$` here with a \, for instance:

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

but quoting the `EOF` terminator is a more reliable way of achieving the same effect.

Catching Interrupts with `onintr`

If your script creates temporary files, you can use `onintr` to catch interrupts, so that the script can delete them before halting.

`onintr` *label*

where *label* is a label in your program that is followed by your housekeeping commands. If the C-Shell receives an interrupt, it performs a `goto` *label*, and executes those commands.

Exit

You can also use the `exit` command (which is built in to the C-Shell) to terminate the script. If you wish to exit with a nonzero status, do the following:

`exit (` *status* `)`

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

C

C-Shell Builtin Commands

C

C-Shell Builtin Commands

`alias` [*name* [*expansion*]]	`alias` with no arguments prints the list of aliases and their expansions. With the *name* argument, `alias` prints the expansion for the named alias if one exists. `alias` with both arguments assigns an expansion to an alias name. For reliability, enclose the expansion in single quotes, and precede exclamation points within it with backslashes (\\).
`bg` [`%`*job* ...]	Puts the current or the specified jobs into the background, resuming execution if they were stopped.
`break`	Resumes execution after the `end` of the nearest enclosing `foreach` or `while` loop. The remaining commands on the current line are executed. Multi-level breaks are produced by writing several `break` commands on one line.
`case` *label:*	A label in a `switch` statement as discussed below.
`cd` [*name*] `chdir` [*name*]	Changes the working directory to *name*. If no *name* is given, either command changes to your home directory. If *name* is not found as a current subdirectory, and is not a pathname beginning with `/`, `./` or `../`, each component of the variable `cdpath` is checked to see if it has a subdirectory *name*. Finally, if *name* is a shell variable with a value beginning with `/`, that value is used as a pathname for the directory to change to.
`continue`	Continue execution of the nearest enclosing `foreach` or `while` loop. The rest of the commands on the current line are executed.
`dirs`	Prints the directory stack, with the top of the stack at the left. The leftmost entry in the stack is the current directory.
`echo`, [`-n` [*words*...]]	The specified *words* are written to the shell's standard output, separated by spaces, and terminated with a newline unless the `-n` option is specified.
`eval` *arg* ...	The arguments are read as input to the shell, which executes the resulting command(s). This is usually used to execute commands generated as a result of command or variable substitution, since parsing occurs before these substitutions. `eval` is often used with `tset`.

`exec` *command*

The specified command is executed in place of the current shell.

`exit` [*code*]

With no arguments the shell exits with the value of the `status` variable. With one argument, shell exits with the specified *code*.

`fg` [`%`*job* ...]

Brings the current *job* or specified *job* s into the foreground, resuming execution if they were stopped.

`foreach` *name* (*wordlist*)
commands
`end`

`foreach` performs a set of *commands* for each item in a list of words. The variable *name* is successively set to each member of *wordlist*. The `foreach` and the `end` commands must appear alone on a line.

The builtin command `continue` halts the current iteration and resumes performing the loop with the next word in the list. When encountered, `break` terminates the entire loop.

When typed in from the terminal, `foreach` prompts for commands until you type `end` on a line by itself; it then performs the loop.

`glob` *wordlist*

Like `echo` but no '\' escapes are recognized. Words are delimited by null characters in the output. This is useful for programs that use the C-shell filename expansion to process a list of words.

`goto` *word*

The specified *word* is filename and command expanded to yield a *label*. The shell then searches (backward, then forward within the script) for a line of the form '`label`: *label*' (possibly preceded by blanks or tabs). Execution continues after the line located by the search.

`hashstat`

Display statistics to indicate how effective the internal hash table has been at locating commands (and avoiding `exec's).` An `exec` is attempted for each component of the `path` where the hash function indicates a possible hit, and in each component that does not begin with a /.

`history` [`-rh`] [*n*]

Displays the history event list, oldest first; when specified, only the *n* most recent events are shown.

`-r` reverses the order of printout to be most recent first.

`-h` displays the history list without leading numbers, (useful for producing scripts on the fly).

`if` (*expr*) *command* [*arg* ...]

If the *expr* evaluates to true, the C-shell performs the *command* with its arguments (*arg* ...). A *command* must be a simple command, not a pipeline, command list, or parenthesized command list. Note that output and input redirection occurs even if the *expr* is false and the *command* is not performed.

`if (`*expr*`) then`
commands
`[else] ∣`
`[else if (`*expr*`) then]`
commands
`endif`
	If the specified *expr* is true, the C-shell performs the command lines between the first `then` statement and the `else` or `else if` statement. Otherwise, commands between the `else` or `else if` statement and the `endif` statement are performed. When an `else if` statement is encountered, its *expr* argument is evaluated. Any number of `else if` branches can be nested. The entire sequence within the `if` block is terminated with one `endif` statement.
	
	The words `else` and `endif` must appear at the beginning of input lines; `if` must appear alone on its input line (or after an `else`.)

`jobs [-l]`	Lists the active jobs; with the `-l` option, lists process id's in addition to the normal information.
`kill [`*-signal*`] %`*job...* `kill [`*-signal*`]` *pid...* `kill -l`	Sends either the TERM (terminate) signal or the specified *signal* to the specified *job(s)* or process id(s) (*pid*). A *signal* can be specified either by number or by name (as shown in `/usr/include/signal.h`, and stripped of the prefix `SIG`). *signal* names are listed by `kill -l`. There is no default, typing '`kill`' does not send a signal to the current job. If the signal being sent is `TERM` (terminate) or `HUP` (hangup), the job or process is sent a `CONT` (continue) signal as well.
`label:`*label*	*Create a label named* `label`*, for use with the* `goto` *command.*
`limit [`*resource*` [`*max_use*`]]`	Limits consumption by the current process and its child processes, each to within *max_use* on the specified *resource.* If no *max_use* is given, the current limit is printed; if no *resource* is given, all limitations are given. *resources* include: `cputime` — the maximum number of cpu-seconds to be used by each process `filesize` — the largest single file which can be created `datasize` — the maximum growth of the `data+stack` region beyond the end of the program text `stacksize` — the maximum size of the automatically-extended stack region, and `coredumpsize` the size of the largest core dump that will be created. *max_use* may be given as a number followed by a scale factor. For all limits other than `cputime` the default scale is `k` or 'kilobytes' (1024 bytes); you can specify `m` or 'megabytes' as a scaling factor. For `cputime` the default scaling is `seconds`, but you can specify `m` for minutes, `h` for hours, or a limit of the form *mm*`:`*ss* for minutes and seconds.
`login`	Terminate a login shell and run `/bin/login` instead.
`logout`	Exit from a login shell.
`nice [+`*n*`] [`*command* `[`*arg* ...`]]`	

`nice` with no arguments sets the C-shell's priority to 4. With a *+n* argument, `nice` sets the priority to *n*. The superuser (root) can set a negative priority by replacing the plus sign with a minus (–). When followed by a simple *command* and its arguments, `nice` sets the priority for that *command*, which is performed by a subshell.

`nohup` [*command* [*arg* ...]]

When supplied with a simple *command* and its arguments, `nohup` runs the command and ignores hangup signals. That is, the command continues to be performed even after you have logged out. Processes that run in the background ignore hangup signals automatically.

`notify %`*job* ...

Notify the user when the status changes for the current job or specified job(s), normally before a prompt. This is automatic when you set the `notify` C-shell variable.

`onintr` [– | *label*]

With a minus-sign (–), the C-shell or script ignores interrupts. With a *label*, the C-shell or script executes a `goto` *label* statement when an interrupt occurs. With no arguments, `onintr` restores the default behavior; the C-shell terminates scripts on an interrupt.

If a C-shell is running detached, with interrupts ignored, *onintr* is also ignored; interrupts continue to be ignored by that C-shell and its commands.

`popd` [*+n*]

Pops the directory stack, returning to the new top directory. With an argument '*+n*', it discards the *n*th stack entry. The elements of the directory stack are numbered from 0 starting at the top.

`pushd` [*name* | *+n*]

With no arguments, `pushd` exchanges the top two elements of the directory stack. Given a *name* argument, `pushd` changes to the new directory (as in `cd`) and pushes the old current working directory onto the directory stack. With a numeric argument '*+n*', it rotates the *n*'th entry in the directory stack to the top and changes to it. Entries in the directory stack are numbered from the top starting at 0.

`rehash`

Recompute the internal hash table of directories in the `path` variable. This is needed when new commands are added to directories in the `path` while you are logged in. It should only be necessary if you add commands to one of your own directories, or if the contents of one of the system directories changes.

`repeat` *count command* [*arg* ...]

Performs the specified simple *command count* times. I/O redirections occur exactly once, even if *count* is 0.

`set` [*name*]
`set` *name=value*
`set` *name*[*index*]=*value*
`set` *name*=(*word_list*)

With no arguments, `set` displays the value of all shell variables. Variables which have other than a single word as value are displayed as a parenthesized *word_list*. With a *name=value* construct, the variable is set to the specified *value*. The third form changes the *index*'th component of variable to *value* if that component exists. The fourth form sets *name* to the list of values in *word_list*. In all cases, the value is command and filename expanded. Variable expansion happens for all arguments before any new values are set.

`setenv` *name value*	Sets the value of environment variable *name* to be *value*, a single string. The most commonly used environment variables, `USER`, `TERM`, and `PATH`, are automatically imported to and exported from the `csh` variables `user`, `term`, and `path`, respectively.
`shift` [*variable*]	The components of the named *variable* are shifted to the left; the leftmost component is discarded. When no *variable* is specified, the arguments list `argv` is shifted left. An error results if `argv`, or the named *variable*, is not set, or has less than one word as a value.
`source` [`-h`] *filename*	The shell reads commands from *filename*. `source` commands can be nested, but if nested too deeply, the C-shell may run out of file descriptors. An error in a `source` file terminates all nested `source` commands. Normally, input during `source` commands is not placed on the history list; the `-h` option places the commands in the history list without being executed.
`stop %`*job* ...	Stops the current or specified background job(s).
`suspend`	Stops the C-shell, as if it received a stop signal [CONTROL-Z]; most often used to stop C-shells started by `su`.

`switch` (*string*)
`case` *label*:
commands
`breaksw`
commands
`default:`
commands
`breaksw`
`endsw`
	The `switch` command begins a multiple branch. *string* is command and filename expanded, and then compared with each label until a match is found. `case` and `default:` statements must begin a line. The file metacharacters `*`, `?` and `[...]` can be used in labels, which are variable expanded. The C-shell executes the block of commands between the matching `case` and the subsequent `breaksw`. If no match is found before a `default:` statement is reached, the C-shell executes the commands between it and the subsequent `breaksw`. When `breaksw` is reached, execution continues after the `endsw`. Without a `breaksw`, commands within subsequent `case` and `default` blocks are performed.

`time` [*command* [*arg* ...]]	With no argument, a summary of the time used by the C-shell and its children is printed. If given, the specified simple *command* is executed and timed. If necessary, a subshell is created to print the time summary for *command*.
`umask` [*nnn*]	With no arguments, `umask` displays the permission-modes *user mask* in octal digits. The user mask is deducted (by a logical *not and*) from the default permission mode setting (666 for files, and 777 for directories) to yield the permissions for newly created files or directories. The default `umask` value is `022`, resulting in permission modes of `644` for new files, and `755` for new directories. With an argument, `umask` sets a new value for the user mask (and resulting permissions). For instance, the command `umask 002` yields permission modes of `664`, for new files (allowing read and write access to the owner and the group, and read-only access to others), and `775` for new

	directories (allowing read, modify and search access to the owner and the group, and read and search access to others).
`unalias` *pattern*	All aliases with names that match the specified *pattern* are discarded. Thus all aliases are removed by `unalias *`.
`unhash`	Use of the internal hash table to speed location of executed programs is disabled.
`unlimit` [*resource*]	Removes limitations on *resource*. If no *resource* is specified, all *resource* limitations are removed.
`unset` *pattern*	All variables with names matching the specified *pattern* are removed. All variables are removed by `unset *`; this includes `home`, `path` and `user`, and so should be avoided.
`unsetenv` *pattern*	Removes all environment variables with names matching the specified pattern.
`wait`	The C-shell awaits all background jobs. If the C-shell is interactive, an interrupt can disrupt the `wait`. When interrupted, the shell prints names and job numbers of all outstanding jobs.
`while` (*expr*) *commands* `end`	Performs the commands between the `while` statement and the `end` statement as long as *expr* evaluates non-zero. `break` and `continue` can be used to terminate or continue the loop prematurely. The `while` and `end` statements must appear alone on the input line. If the input source is a terminal, the C-shell prompts for commands during the first pass through the loop. Subsequent passes are performed without prompting.
`%`[*job*]	Brings the current or specified *job* into the foreground.
`%`[*job*] `&`	Continues the current or specified job in the background.
`@` [*variable*=*expr*] `@` [*variable* [*index*] =*expr*]	With no arguments, `@` prints the values of all the shell variables. With arguments, `@` sets the specified *variable* to the value of *expr*, or the *index*'th component of that variable to *expr*. If *expr* contains <, >, & or | then it must be placed within parentheses. Both *variable* and its *index*'th component must already exist. The operators *=, +=, etc., are available as in C. When given arguments, there must be a space after the '@' sign. The space separating the variable from the assignment operator is optional. Spaces must separate components of *expr*. Special postfix ++ and -- operators increment and decrement *variable* respectively, that is, `@ i++` increments `i` by one.

D

C-Shell Special Characters

D

C-Shell Special Characters

Characters with special meaning to the C-Shell:

`?`	Single character wild card.
`*`	String wild card, zero or more characters.
`.`	Abbreviation for current working directory.
`..`	Abbreviation for the parent of the current directory.
`~`	Abbreviation for your home directory.
`~`*user*	Abbreviation for the home directory of *user*.
`[...]`	Matches any single character listed within the brackets.
[*x*−*y*]	Matches any character within the range of *x* and *y*.
{*str*, ...}	*Grouping*. Matches each *str* successively. Filename substitution is applied to each *str* before matching occurs. Thus, `{x,*y*,?z*}` matches a filename `x`, all filenames containing the letter `y`, and all filenames having `z` as the second character. Groups enclosed with braces can be nested.
`&`	Places the command in the background.
CTRL-Z	Stops the foreground job, placing it in the background.
`%`[*n*]	Brings the current (stopped) job, or the specified background job to the foreground.
`%`[*n*] `&`	Continues, in the background, the current or specified stopped job.
>*filename*	Redirects the standard output to *filename*. If *filename* already exists, its previous contents are lost. When set, the shell variable `noclobber` prevents redirection to existing files or character special devices.
>! *filename*	Forces the standard output to *filename*, even when `noclobber` is set.

`>&` *filename*
Routes diagnostic (standard error) output to *filename*, along with the standard output.

`>&!` *filename*
Forces diagnostic and standard output to *filename*.

`>>` *filename*
Appends the standard output to *filename*. When `noclobber` is set, the file must already exist.

`>>!` *filename*
Forces the standard output to *filename*, even when `noclobber` is set. Creates a new file if necessary.

`>>&` *filename*
Appends the diagnostic as well as standard output to *filename*.

`>>&!` *filename*
Forces appending of diagnostic and standard output to *filename*, even when `noclobber` is set.

cmd `|` *cmd*
Pipe. Uses the standard output of the left-hand *cmd* as standard input for the right-hand *cmd*.

cmd `|&` *cmd*
Uses both standard and diagnostic output of the left-hand *cmd* as standard input for the right-hand *cmd*.

`(...)` *Command grouping*. Commands and pipelines surrounded by parentheses are executed in a subshell and treated as a unit by the current C-Shell.

`(...) >&` *filename*
Redirects the standard output (if any) and the diagnostic output of the enclosed command(s) to *filename*. This is espcially useful if the enclosed commands redirect the stadard output to a file (thus sending the standard output and the standard error to separate destinations).

`<` *filename*
Opens *filename* as the standard input.

cmd `<<` *word*
Here document. Indicates that a command (typically interactive) is to accept *its* commands from the same device or file (usually a script) as the shell. *word* is interpreted literally as the *end-of-input* mark for the command. The C-Shell parses, but does not execute, each text line between the here document and a line containing *word* by itself. After applying command, filename, and variable substitution, the C-shell passes each line on to *cmd*. To suppress all substitution, include a \, ", or ´ in *word*.

;	Separates commands on one input line.
\	At the end of a line, escapes the newline character and continues the command to the next input line.
\	Escape the special meaning of the character it precedes.
´...´	The C-Shell treats the enclosed text as one word, preventing history and variable substitution.
"..."	The C-Shell treats the enclosed text as one word, breaking words only at enclosed newlines. History and variable substitution is performed *before* escape characters are interpreted.
` *command* `	Replaces the backquoted command or pipeline (including the backquote marks) with its output. Output is broken into words at blanks, tabs and newlines, except for the final newline. Unless the right-hand backqoute is followed by a space, the last word of the substitution is is prepended to the following word on the command line.

Escaped history substitution event designators and word designators (described below) can be used to indicate command line arguments within an alias definition.

^*l*^*r*[^]	Substitutes the string *r* for the string *l* in the previous command line. The final ^ is required only if history substitution modifiers are appended.
!	Begins a history substitution. To escape its special meaning, precede the ! with a backslash (\). A ! is also escaped when followed by a blank, tab, newline, (or =.

The following designators select an event (command line) from the history list. Word designators and modifiers can be appended for command-line editing.

!!	The previous command.
!*n*	Command line number *n*.
!–*n*	Selects the event whose number is *n* less than the current one.
!*str*	The most recent command beginning with *str*.
!?*str*[?]	The most recent command containing *str*. The closing question mark is only required when word designators or modifiers are appended.
!*	All arguments from the previous command, but not argument zero (the command name).
!^	The first *argument* from the previous command. If, for instance, the command was `echo first`, then `!^` would expand to `first`.

!$	The last argument from the previous command.
!:*n*	The *n*'th argument from the previous command.
!#	The contents of the *current* command line typed in so far.
!{*str***}**...	Restrict the event designation to *str*; text following the brackets is appended to the last word of the expansion *after* substitution takes place.

Word designators can be appended to the history substitution character (! for the previous event, to a quick substitution, or to an event designator.

:*	All arguments, except argument zero.
:^	The first argument .
:$	The last argument.
:*n*	The *n*'th argument.
:%	The word matched by most recent !? search.
:*x−y*	Argument *x* through argument *y*.
:−*y*	abbreviates **:0−***y*.
:*x******	Argument *x* through the last argument.
:*x***−**	Argument *x* through the next-to-last argument.
:#	The contents of the *current* command line typed in so far.

The following modifiers can be used in any sequence to modify a selected event or word. A colon is required to separate modifier(s) from event or word designators.

[:]p	Prints the new command but does not execute it.
[:]h	Removes a trailing pathname component, leaving the head.
[:]t	Removes all leading pathname components, leaving the tail.
[:]r	Removes a filename extension (.*xxx*).
[:]e	Removes all but the extension.
[:]s/*l*/*r*/	Substitutes *r* for *l*. *l* is a literal string, not a regular expression. Any character may be used as the delimiter in place of /. The character & in the right hand side is replaced by the left hand string. A null *l* uses the previous string either from a *l* or from a ? event search.
[:]&	Repeats the previous substitution.
[:]q	Quotes the substituted words, preventing further substitutions.
[:]x	Like :q, but breaks words at blanks, tabs and newlines.
:g*m*...	*Global prefix*. When prefixed any of the above modifiers, *m*, the modifier(s) apply to all words in the specified event. Normally, each

word must be modified separately.

After the input line is aliased and parsed, and before each command is executed, the C-Shell performs variable substitution on words that start with an unescaped `$`, according to the list below. A `$` is escaped by preceding it with a backslash (\), or when followed by a blank, tab, or end-of-line.

Shell variables have names consisting of up to 20 letters, digits and underscore characters, starting with a letter.

Environment variables can be expanded but not modified.

$*var*	Is replaced with the value of *var*.
${*var***}** ...	The brackets indicate that the enclosed string is the variable name. The value of the named variable is prepended to the text that follows on the command line.
${*var***[***selector***]}**	Select words from within *var*. *selector* can be one of:

n	a number.
x–*y*	two numbers separated by a – to specify a range.
x–	Word *x* through the last word.
–*y*	The first word through word *y*.
*	all words in the value.
$*var*	the value of another variable, in which case variable substitution is applied to the *selector* first, and then to the entire word.

$#*var*	The number of words in the variable.
${#*var***}**	Same as `$#`*var*
$0	The name of the file from which command input is being read. An error occurs if the name is not known.
$*n*	The *n*'th word in the argument list; equivalent to `$argv[`*n*`]`. `Same as $`*n*.
${*n***}**	
$*	All words in the argument list; equivalent to `$argv[*]`.
$?*var*	
${?*var***}**	replaced with `1` if *var* is set, or `0` if not.
$?0	replaced with `1` if the current input filename is known, `0`, otherwise.
$$	Is replaced with the process ID (PID) of the (parent) shell.
$<	replaced with text taken from the standard input, with no further interpretation. Used to read from the keyboard in a C-Shell script.

The modifiers `[:]h`, `[:]t`, `[:]r`, `[:]q`, and `[:]x` can be applied to the substitutions above. See *Modifiers* under *History Substution*, above, for a description.

If braces `{ ... }` appear in the variable substitution, modifiers must be enclosed within them.

The current implementation allows only one modifier within each variable substitution.

The following variable substitutions can not be modified: **$?**, **$$**, and **$<**.

Expressions appear within the `@`, `exit`, `if`, and `while` builtin commands.

Null or missing terms are interpreted as `0`.

Results of all expressions are *strings* that represent decimal numbers. Results of logical expressions are `1` (for true) or `0` (for false).

(...)	Parentheses indicate grouping of operaters and terms within an expression, overriding the standard precedence of operators.
==	True if the string on the left is equal to the string on the right (after all substitutions are performed).
!=	True if the string on the left is not equal to the string on the right.
=~	True if the string on the left is matched by the pattern on the right.
!~	True if the string on the left is not matched by the pattern on the right.
<	True if the number on the left is less than the number on the right.
<=	True if the number on the left is less than or equal to the number on the right.
>	True if the number on the left is greater than the number on the right.
>=	True if the number on the left is greater than or equal to the number on the right.
||	Logical *or* connective.
&&	Logical *and* connective.
{...}	*Command successful*. True if the command surrounded by brackets exits with status code `0`.

An operator of the form

flag filename

is true if the attribute *flag* applies to *filename*, with respect to the current user. *flag* can be one of:

-r	read access
-w	write access
-x	execute access

`-e`	existence
`-o`	ownership
`-z`	zero size
`-f`	plain file
`-d`	directory
! *flag*	true if *flag* does *not* apply.

If the file does not exist, or is inaccessible, then all inquiries yield false as a result.

+	Addition.
–	Subtraction.
*	Multiplication.
/	Division.
%	Remainder after division.
0*str*	A string with a leading zero is interpreted as an octal numeral.
<<	Bitwise *shift left* operator.
>>	Bitwise *shift right* operator.
|	Bitwise *or* operator.
^	Bitwise *exclusive or* operator.
&	Bitwise *and* operator.

E

C-Shell Predefined Variables

E

C-Shell Predefined Variables

`argv`	`argv`, the arguments list, contains any arguments to the shell. The C-Shell uses values from this variable to replace positional parameters. For instance, the C-shell replaces `$1` with `$argv[1]`.
`cdpath`	A list of alternate directories to search within when performing a `chdir` command.
`cwd`	The full pathname of the current directory.
`echo`	Set when the −x command line option is given. Echoes each command and its arguments prior to execution. For nonbuiltin commands all expansions occur before echoing. The C-Shell echos commands before any command or filename substitutions take place.
`histchars`	A string value of two characters that replace those that indicate history substitution. The first character replaces ! as the history substitution character. The second character replaces ^ for quick substitutions.
`history`	A numeric value to control the size of the history list. Commands invoked within the specified number remain in the list. The most recent command is always saved, even when this variable is not set. If you use too large a value for `history`, the shell may run out of memory.
`home`	Your home directory (initialized from the environment). The filename expansion of ~ refers to the value of this variable.
`ignoreeof`	If set, the shell ignores end-of-file signals from the terminal. This prevents the shell from being killed by an accidental CTRL-D.
`mail`	A list of files where the shell checks for mail. If numeric, the first word of the value specifies a mail checking interval in seconds. The default interval is 5 minutes. If multiple mail files are specified, the shell displays

```
New mail in filename
```

for any file with new mail.

`noclobber`	When set, `noclobber` places restrictions on output redirection to insure that files are not accidentally overwritten, and that >> redirections append to existing files.
`noglob`	If set, filename expansion is inhibited. Useful in shell scripts that do not deal with file lists, or after a list of files has been obtained and further expansion is not desired.
`nonomatch`	If set, a file list that, when expanded, does not match any existing files, returns the pattern, rather than an error message.
`notify`	If set, the shell notifies you immediately when jobs are completed, rather than just before printing the next prompt.
`path`	Each word of the value specifies a directory within which to search for named commands (that aren't pathnames). A null value specifies the current directory. If `path` is undefined, only complete pathnames are recognized as commands. Normally, the search path includes the directories `.`, `/usr/bin`, `/usr/ucb`, default search path is `/etc`, `/usr/etc`, `/bin` and `/usr/bin`. A shell that is given neither the `-c` nor the `-t` options will normally hash the contents of the directories in the `path` variable after reading `.cshrc`, and each time the `path` variable is reset. When new commands are added to these directories, it may be necessary to give the `rehash` command for them to be found.
`prompt`	A string to be printed when the shell is ready to accept a command from the terminal. A `!` is replaced by the current event number unless preceded with a backslash (\\).
`savehist`	A numeric value to control the size of the history list saved in `~/.history` whenever you log out. Commands invoked within the specified number are saved. During startup, the shell adds the contents of this file onto the history list so that history can be saved between sessions. Very large values of `savehist` slow down startup of the C-shell.
`shell`	The file in which the shell resides, used to fork shells to interpret scripts.
`status`	The status returned by the last command. If the command terminated abnormally, then 0200 is added to the status. Builtin commands that fail return exit status of `1`; successful builtin commands return `0`.
`time`	Controls automatic timing of commands. The `time` variable can be supplied with one or two values, such as `set time=3` or `set time=(3 "%E`

`%P%")`. The first value is a numeric threshold in seconds of CPU time. The C-shell displays a resource-usage summary for any command running longer than the specified threshold. The second value is optional and is a character string which determines which resources the user wishes displayed. The character string can be any string of text with embedded control key-letters in it. A control key-letter is a percent sign (`%`) followed by a single *upper-case* letter. To print a percent sign, use two percent signs in a row. Unrecognized key-letters are simply printed. The control key-letters are:

`D`	Average amount of unshared data space used in Kilobytes.
`E`	Elapsed (wall clock) time for the command.
`F`	Page faults.
`I`	Number of block input operations.
`K`	Average amount of unshared stack space used in Kilobytes.
`M`	Maximum real memory used during execution of the process.
`O`	Number of block output operations.
`P`	Total CPU time — U (user) plus S (system) — as a percentage of E (elapsed) time.
`S`	Number of seconds of CPU time consumed by the kernel on behalf of the user's process.
`U`	Number of seconds of CPU time devoted to the user's process.
`W`	Number of swaps.
`X`	Average amount of shared memory used in Kilobytes.

The default resource-usage summary is a line of the form:

uuu.u`u` *sss.s*`s` *ee:ee* *pp*`%` *xxx*+*ddd*`k` *iii*+*ooo*`io` *mmm*`pf`+*ww*`w`

where *uuu.u* is the user time `U`, *sss.s* is the system time `S`, *ee* :*ee* is the elapsed time `E`, *pp* is the percentage of CPU time versus elapsed time `P`, *xxx* is the average shared memory in Kilobytes `X`, *ddd* is the average unshared data space in Kilobytes `D`, *iii* and *ooo* are the number of block input and output operations respectively `I` and `O`, *mmm* is the number of page faults `F`, and *ww* is the number of swaps `W`.

`verbose`

When set by the −v command line option, the C-shell displays the words of each command after history substitution and before execution.

F

Bourne Shell Scripts

F

Bourne Shell Scripts

You can use the Boure Shell to perform a set of UNIX commands contained in a file called a *script*.

Bourne Shell scripts do not serve the same function as `make`, which is useful for consistently performing a set of operations on related files. While scripts can be written to do this, the Bourne Shell is more general in scope. Scripts do not check for dependencies, for instance. And, there are many things that you can do with scripts, such as prompting for input from the terminal, that are not practical using `make`.

To run a Bourne Shell script (for which you have execute permission), type in its filename as if it were a command. When you do, the system looks at the very first line of the file to decide which Shell should run the script:

- If the first line does *not* start with a # (hash sign), the system uses the Bourne Shell to run the script.
- If the first line starts with a # (hash sign) and is *not* followed by a `!` (exclamation mark), the system uses the C-Shell to run the script.
- Finally, if the first line of the Shell script starts with a `#!` combination and is followed immediately by a name, the system looks for a program of that name to run the Shell script. If you supply arguments on the command line, these are passed along to variables in the Bourne Shell called *positional parameters*. The first argument after the name of the script is placed in variable `1`. The second is placed in variable `2`, and so forth.

NOTE *You can often simplify testing of Bourne Shell scripts (or commands to run within them) by using the Bourne Shell interactively. To do so, type in the command* `/bin/sh`, *and enter commands as described in this Appendix. Use* [CTRL-D] *to exit and return to the C-Shell. Most of the examples below make use of the Bourne Shell interactively, as well as within scripts.*

Bourne Shell Variables

The Bourne Shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. You may assign values to variables by writing the variables name, an equal sign, and a value (with no spaces between). For example:

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

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

```
$ cheese=
$ cheese=
```

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

```
mars% cat > test
echo $user
^D
mars% chmod 755 test
mars% test
fred
mars%
```

You can use variables to provide abbreviations for strings that are used frequently throughout a script. A script containing the following lines

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

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

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed immediately by a letter or digit:

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

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

```
ps a >$tmpa
```

redirects it to a file named *tmpa.*

Bourne Shell Initial Variables

Except for `$?`, the variables defined in table are set initially by the Bourne Shell. `$?` is set after executing each command.

Table F-1 *Variables Initialized by the Bourne Shell*

Variable	*Explanation*
`$?`	The exit status (return code) of the last command executed, as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned.
`$#`	The number of positional parameters (in decimal).
`$$`	The process number of this Shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary filenames.
`$!`	The process number of the last process run in the background (in decimal).
`$-`	The current Bourne Shell flags, such as `-x` and `-v` .

Variables with Special Meaning to the Bourne Shell

Some variables have a special meaning to the Bourne Shell; avoid them in general use.

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

The file `.profile` in your home directory is the setup file for the Bourne Shell — equivalent to the combination of the `.cshrc` and `.login` files for the C-Shell.

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

`$HOME` Your home directory; this variable is also typically set in `.profile.`

`$PATH` A list of directories that contain commands (the *search path*). Each time the Bourne Shell executes a command, a list of directories is searched for an executable file by that name. If `PATH` is not set, then the current directory, `/bin`, and `/usr/bin` are searched by default. `$PATH` consists of directory names separated by `:`. For example,

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

specifies that the current directory (the null string before the first `:`), `/usr/fred/bin`, `/bin`, and `/usr/bin` are to be searched in that order. This allows you to have your own private commands accessible independently of the current directory. If the command name contains a `/`, then this directory search is not used.

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

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

`$IFS` The set of characters to be interpreted as blanks when parsing command lines.

The test Command

Although the `test` command is not part of the Bourne Shell, scripts frequently use it. `text` can be used to check on the status of files, to compare strings and algebraic expressions, and to perform integer calculations. For instance:

```
test -f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Here is the list of things you can test for.

`-b` *file*	true if *file* exists and is a block special device.
`-c` *file*	true if *file* exists and is a character special device.
`-d` *file*	true if *file* exists exists and is a directory.
`-f` *file*	true if *file* exists and is not a directory.
`-g` *file*	true if *file* exists and is setgid.
`-h` *file*	true if *file* exists and is a symbolic link.
`-k` *file*	true if *file* exists and is sticky.
`-l` *string*	the length of *string*.
`-n` *string*	true if the length of *string* is nonzero.
`-r` *file*	true if *file* exists and is readable.
`-s` *file*	true if *file* exists and has a size greater than zero.
`-t` [*fildes*]	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
`-w` *file*	true if *file* exists and is writable.
`-x` *file*	true if *file* exists and is executable.
`-z` *string*	true if the length of *string* is zero.
string-1 = *string-2*	true if the strings *string-1* and *string-2* are equal.
string-1 != *string-2*	true if the strings *string-1* and *string-2* are not equal.
string	true if *string* is not the null string.
n1 `-eq` *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons `-ne`, `-gt`, `-ge`, `-lt`, or `-le` may be used in place of `-eq`.

`[ ... ]` *alternative form of the* `test` *command*

You can also call `test` by surrounding the expression to be tested with brackets (`[ ]`). (The left bracket is a command name, the right bracket is a argument signifying the end of the expression.) This form is most often used with the `if` command described later on.

Getting Started — A Simple Procedure

Here is a vary simple Bourne Shell procedure to look up names in a list of names and telephone numbers contained in a file called `names.list`. Let's call the lookup procedure `name`:

```
$ cat name
#! /bin/sh
grep -i $1 names.list
$
```

This is about as simple as you can get. Let's run the `name` procedure looking for people called *Tom* something-or-other:

```
$ name tom
Tom Athanasiou          toma@thales             7534
Tom McReynolds          bohica@centauri         7256
$
```

Later on we will show a more sophisticated version of `name`, and expand on this procedure to demonstrate other features of the Bourne Shell.

Control Flow in the Bourne Shell — `for`

A frequent use of Bourne Shell procedures is to loop through the arguments (`$1, $2, . . .`) executing commands once for each argument. Here's an expanded version of the `name` procedure from above. The original version of `name` can only look for one person's name. Now we want to expand it to look for more than one name at a time. Let's look at the new version:

```
$ cat name
#! /bin/sh
for person
    do grep -i $person names.list
done
$
```

Here we set a variable called `person` to the value of each positional parameter, one at a time, then we call out the value of `person` in the `grep` command. Now we can look for more than one name at a time:

```
$ name bill mary
Bill Tuthill            tut@cairo               7258
Mary Hamilton           hamilton@artemis        7214
$
```

General form of the `for` *loop*

The `for` loop notation is recognized by the Bourne Shell and has the general form

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

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like `do` and `done` are only recognized following a newline or semicolon. *Name* is a Bourne 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.

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

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

The command:

```
$ create alpha beta
```

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

Control Flow in the Bourne Shell — `case`

The `case` notation provides a multi-way branch. For example:

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

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

```
$ append file
```

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

```
$ append file1 file2
$
```

If the number of arguments supplied to *append* is other than 1 or 2, a message is displayed indicating proper usage.

The general form of the `case` command is:

```
case word in
    pattern-1)  command-list-1;;
    pattern-2)  command-list-2;;
    . . .
esac
```

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

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example the commands following the second * are never executed.

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

Another example of the use of the `case` construction is to distinguish between different forms of an argument. The following example is a fragment of a `cc` (C compiler) command:

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

Matching Multiple Patterns in One Case

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 `?`.

Here Documents in the Bourne Shell

Sometimes a Shell procedure requires data. Instead of having the data in some file somewhere in the system, the data can be included as part of the Shell procedure. Such a collection of data is called a *here document* — the data (document) is right *here* in the Shell procedure. One advantage of a here document is that Shell parameters can be substituted in the document as the Shell is reading the data.

The general form of a here document is like this:

```
        lines of Shell commands
        . . .
command-name << end-marker
lines of data
belonging to the
here document
        . . .
end-marker
        . . .
        more lines of Shell commands
```

The name *command using here document*

Let's revisit the name procedure discussed in earlier sections. Instead of having the names and numbers in one file and the Shell procedure in another file, you can keep both the procedure and the list in the same file — that is, in the procedure. Here's another version of the name command:

```
$ cat name
#! /bin/sh
grep -i $1 <<EOF
Tom Athanasiou            toma@thales              7534
Bridget Burke             bridget@sid              7441
    . . .
    more names
    . . .
Daniel Sears              sears@sasha              7435
Bill Tuthill              tut@cairo                7258
Dirk van Nouhuys          dirk@words               7296
EOF
$
```

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

Now you'll notice that in *this* version of name we're back to being able to only look up one name at a time. We *could* combine the multiple-name version with the here-document version:

```
$ cat name
#! /bin/sh
for person
    do grep -i $person <<EOF
Tom Athanasiou          toma@thales             7534
Bridget Burke           bridget@sid             7441
    . . .
    more names
    . . .
Daniel Sears            sears@sasha             7435
Bill Tuthill            tut@cairo               7258
Dirk van Nouhuys        dirk@words              7296
EOF
done
$
```

The problem with this approach is that the Shell reads up the list of names every time around the `for` loop. This could become excruciatingly slow. In a later section we show another version of `name` using temporary files for faster performance.

Parameter substitution in here documents

Parameters are substituted in the here document before it is made available to whatever command as illustrated by the following procedure called `edg` (`ed` globally).

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

Then the command line:

```
$ edg string1 string2 file
```

is equivalent to the command:

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

and changes all occurrences of *string1* in *file* to *string2*. You can prevent substitution by using '\' to quote the special character `$` as in

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

This version of *edg* is equivalent to the first except that *ed* displays a `?` if there are no occurrences of the string `$1`. Quoting the terminating string prevents substitution entirely within a *here* document, for example:

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

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

Control Flow in the Bourne Shell — `while`

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

```
while command-list-1
do command-list-2
done
```

The value tested by the `while` command is the exit status of the last simple command following `while`. Each time round the loop *command-list-1* is executed; if a zero exit status is returned then *command-list-2* is executed; otherwise, the loop terminates. For example,

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

is equivalent to

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

`shift` is a Bourne Shell command that renames the positional parameters `$2, $3, . . .` as `$1, $2, . . .` and discards `$1`.

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

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

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

Control Flow in the Bourne Shell — `if`

A general conditional branch of the form

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

is also available to test the value returned by the last simple command following `if`.

We can illustrate a very simple use of the `if` command by expanding on our `name` procedure from before. The relevant change is in the first few lines:

```
$ cat name
#! /bin/sh
if test $# -lt 1
then
        echo Usage: $cmd name ...
        exit 1
fi
grep -i $1 <<EOF
Tom Athanasiou          toma@thales             7534
Bridget Burke           bridget@sid             7441
    . . .
    more names
    . . .
Daniel Sears            sears@sasha             7435
Bill Tuthill            tut@cairo               7258
Dirk van Nouhuys        dirk@words              7296
EOF
$
```

The change here is the `if` command — the original version of the procedure didn't check that the user supplied any parameters at all. This version checks the number of parameters (`$#`) using the `test` command, and displays a *usage* message if there are no parameters to remind the user of the correct way to use the procedure.

We mentioned earlier that the `test` command can also be written as `[`. Here is the first couple of lines of the above`name`procedure

```
$ cat name
#! /bin/sh
if [ $# -lt 1 ]; then
        echo Usage: $cmd name ...
        exit 1
fi
grep -i $1 <<EOF
    . . .
EOF
$
```

The `if` command may also 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
```

Here is an example of the `test` command in action. This is an extract from the `diff3` Shell procedure:

```
$ cat -n /usr/bin/diff3
     1  #! /bin/sh
     2  e=
     3  case $1 in
     4  -*)
     5    e=$1
     6    shift;;
     7  esac
     8  if test $# = 3 -a -f $1 -a -f $2 -a -f $3
     9  then
    10    :
    11  else
    12    echo usage: diff3 file1 file2 file3 1>&2
    13    exit
    14  fi
    15  trap "rm -f /tmp/d3[ab]$$" 0 1 2 13 15
    16  diff $1 $3 >/tmp/d3a$$
    17  diff $2 $3 >/tmp/d3b$$
    18  /usr/lib/diff3 $e /tmp/d3[ab]$$ $1 $2 $3
```

The relevant line is on line 8 that reads

```
if test $# = 3 -a -f $1 -a -f $2 -a -f $3
```

This says that if the number of parameters (`$#`) is equal to 3, and all three parameters are files, the procedure can continue, otherwise the procedure displays an error message and stops.

`elif` *multiple-test version of* `if`

A multiple-test `if` command of the form

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

may be written using an extension of the `if` notation:

```
if condition-1
then    actions-1
elif    condition-2
then    actions-2
elif    condition-3
. . .
fi
```

The sequence

```
if command-1
then    command-2
fi
```

may be written

```
command-1 && command-2
```

Conversely,

```
command-1 || command-2
```

executes *command-2* only if *command-1* 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 )
```

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

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

executes `rm junk` in the directory x without changing the current directory of the invoking Shell.

The commands

```
$ cd x; rm junk
$
```

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

Debugging Bourne Shell Procedures

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

```
set -v
```

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

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

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

The command

```
set -x
```

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

```
set -
```

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

Keyword Parameters in the Bourne Shell

Bourne Shell variables may be given values by assignment or when a Shell procedure is invoked. An argument to a Bourne 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
```

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

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

```
set - *
```

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

Parameter Transmission in the Bourne Shell

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

```
export user box
```

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

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

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

Parameter Substitution in the Bourne Shell

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

```
$ echo $d
```

or

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

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

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

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

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

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

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

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

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

which substitutes the same string as

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

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

If there is no sensible default then the notation

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

echos the value of the variable d if it has one; otherwise the Bourne Shell prints *message*, if the Shell if not interactive, and stops executing the procedure. If *message* is absent, then a standard message is printed. A Bourne Shell procedure that requires some parameters to be set might start as follows.

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

Colon (:) is a command that is built in to the Bourne Shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct** or **bin** are not set, and the Shell is not interactive, the Shell stops executing the procedure.

Command Substitution in the Bourne Shell

In a similar way, you can substitute the standard output from a command as the value of a parameter. The command *pwd* displays on its standard output the name of the current directory. For example, if the current directory is */usr/fred/bin* then the command

```
d=`pwd`
```

is equivalent to

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

The entire string between grave accents[50] (`...`) 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 use of string processing commands within Bourne Shell procedures. An example of such a command is *basename,* which removes a specified suffix and the pathname's prefix from a string. For example,

```
basename /usr/fred/main.c .c
```

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

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

that sets `B` to the part of `$A` with the pathname and suffix `.c` stripped.

Here are some composite examples.

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

Evaluation and Quoting in the Bourne Shell

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

[50] Often called backquotes.

Commands are parsed initially according to the grammar given in the 'Grammar' section. Before a command is executed, the following substitutions occur.

- Parameter substitution, such as `$user`
- Command substitution, such as `` `pwd` ``

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

  ```
  echo $X
  ```

  will echo *$y*.

- Blank interpretation

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

  ```
  echo ''
  ```

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

  ```
  echo $null
  ```

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

- Filename generation

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

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

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

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

For example,

```
echo "$x"
```

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

```
echo "$*"
```

passes the positional parameters as a single argument and is equivalent to

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

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

```
echo "$@"
```

passes the positional parameters, unevaluated, to *echo* and is equivalent to

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

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

Table F-2 *Quoting Mechanisms*

Quoting Character	*Metacharacter*					
\	$	*	`	"	´	
´	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

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

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

```
eval echo $X
```

echos the string *pqr*.

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

```
wg=´eval who|grep´
$wg fred
```

is equivalent to

```
who|grep fred
```

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

Error Handling in the Bourne Shell

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

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

- Input/output redirection may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a 'bus error' or 'memory fault.' See table F-3 for a complete list of UNIX signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the Bourne Shell goes on to execute the next command. Except for the last case, the Bourne Shell displays an error message. All remaining errors cause the Bourne Shell to exit from a command procedure. An interactive Bourne Shell will return to read another command from the terminal. Such errors include the following:

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

The Bourne Shell flag `-e` terminates the Bourne Shell if any error is detected.

Table A-3 *UNIX Signals*

Signal Name	*Signal Number*	*Notes*	*Description*
SIGHUP	1		hangup
SIGINT	2		interrupt
SIGQUIT	3	*	quit
SIGILL	4	*	illegal instruction
SIGTRAP	5	*	trace trap
SIGIOT	6	*	IOT instruction
SIGEMT	7	*	EMT instruction
SIGFPE	8	*	floating point exception
SIGKILL	9		kill — cannot be caught, blocked, or ignored
SIGBUS	10	*	bus error
SIGSEGV	11	*	segmentation violation
SIGSYS	12	*	bad argument to system call
SIGPIPE	13		write on a pipe with no one to read it
SIGALRM	14		alarm clock
SIGTERM	15		software termination signal from kill
SIGURG	16		urgent condition on IO channel
SIGSTOP	17	†	stop — cannot be caught, blocked, or ignored
SIGTSTP	18	†	stop signal from tty
SIGCONT	19	•	continue after a stop — cannot be blocked
SIGCHLD	20	•	to parent on child stop or exit
SIGTTIN	21	†	background read attempted from control terminal
SIGTTOU	22	†	background write attempted from control terminal
SIGIO	23		input/output possible signal *
SIGXCPU	24		exceeded CPU time limit
SIGXFSZ	25		exceeded file size limit
SIGVTALRM	26		virtual time alarm
SIGPROF	27		profiling time alarm
SIGWINCH	28	•	window changed

Notes on the Signals

* These signals normally create a memory image of the terminated process.

• These signals are discarded if the signal action is SIG_DFL.

† These signals normally stop the process.

The Bourne Shell itself ignores quit, which is the only external signal that can cause a dump. The signals in this list of potential interest to Bourne Shell programs are 1, 2, 3, 14 and 15.

Fault Handling in the Bourne Shell

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

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

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

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

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

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

The use of *trap* is illustrated by this modified version of the `name` command. You'll recall that the version of the `name` command shown using a *here* document would only look for one name at a time and that if we modified it to look for multiple names, the *here* document would be read every time around the `for` loop. Here is a version that copies the *here* document into a temporary file. The name of the temporary file is derived from the process ID of this command. When the procedure terminates, the `trap` is called to remove the temporary file. Let's take a look at this version of the `name` command:

```
#! /bin/sh -u
if [ $# -lt 1 ]; then
    echo Usage: name person ...
    exit 1
fi
junk=/tmp/$cmd.$$
trap "rm -f $junk; exit" 0 1 2 15
cat > $junk <<EOF
Tom Athanasiou          toma@thales             7534
Bridget Burke           bridget@sid             7441
    . . .
    more names
    . . .
Daniel Sears            sears@sasha             7435
Bill Tuthill            tut@cairo               7258
Dirk van Nouhuys        dirk@words              7296
EOF
for person
    do grep -i $person $junk
done
```

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

Since there is no signal 0 in UNIX, the Bourne Shell uses it to indicate the commands to be executed on exit from the Bourne Shell procedure.

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

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

which causes both the procedure and the invoked commands to ignore the *hangup*, *interrupt*,and *kill* signals.

Traps may be reset by saying:

```
trap 2 3
```

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

```
trap
```

The `scan` *Command*

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

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

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

Here is an example of the `scan` command in action:

```
$ scan
bin:
ls
diffmark     henry.pct    lifescreen    scan.sh
bin:
^D
experiments:
ls
Makefile        doctools        macro.packages  test.bs
Old.Stuff       ellipse.ps      macros          test.pages
diffs           junk            new.macros      tmac.ex
experiments:
rm junk
experiments:
^D
misc:
ls -CF
addresses/       memos/           squash/
henry.raving/    quotes/          status.reports/
howto/           ski.cabins/      stoneman/
jokes/           solar/           sugfest/
letters/         sources/         sun.board
misc:
^D
system.v.book:
ls
Makefile            intro.mexp          shell.mexp
book.mss            login.mexp          shex1.mss
docprep.mexp        mail.mexp           shex2.mss
ed.and.sed.mexp     manpage.mss         softtool.mexp
ex.mexp             misc                stdio.mexp
filesystem.mexp     preface.mexp        system.admin.mexp
headex.mss          roman.mss           tablex.mss
system.v.book:
^D
$
```

Command Execution in the Bourne Shell

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

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

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

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

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

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

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

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

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

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

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

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

The environment for a command run in the background such as

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

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the Shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
$ ed file &
```

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

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

Calling the Bourne Shell

The Bourne Shell interprets the following flags when it is called. If the first character of argument zero is a minus, then commands are read from the file *.profile*.

`-c` *string*
: If the `-c` flag is present, commands are read from *string*.

`-s` If the `-s` flag is present or if no arguments remain, commands are read from the standard input. Bourne Shell output is written to file descriptor 2.

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

Bourne Shell Grammar

Commands are parsed initially according to the following grammar.

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

simple-command: item
      simple-command item

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

pipeline:        command
      pipeline | command

andor:           pipeline
      andor && pipeline
      andor || pipeline
```

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

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

file: *word*
& *digit*
& -

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

pattern: *word*
pattern | *word*

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

empty:

word: *a sequence of non-blank characters*

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

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

Bourne Shell Metacharacters and Reserved Words

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

Patterns	`*`	match any character(s) including none
	`?`	match any single character
	`[...]`	match any of the enclosed characters
Substitution	`${...}`	substitute Shell variable
	`` `...` ``	substitute command output
Quoting	`\`	quote the next character
	`'...'`	quote the enclosed characters except for ´
	`"..."`	quote the enclosed characters except for `$` `` ` `` `\` `"`

Reserved Words

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

G

Command Summary

G

Command Summary

Filename Substitution

`[`*range*`]`
Match characters in a list or range.

`[ab]*`
matches filenames starting with `a` or `b`.

`[a-Z1-0]*`
matches filenames starting with any alphanumeric character.

`{`*string*`,` *string*`}`
Match enclosed strings.

`{venus,mars}`
matches the filenames `venus` and `mars`.

File Properties

`chmod` *arg filename*
change permissions. *arg* is one of:

ddd
where *d* is a digit from `0` to `7` .

class op perm ...
where *class* , *op* and *perm* , are taken from:

class		*op*		*perm*	
u	user (owner)	=	set permission	r	read
g	group	-	remove access	w	write
o	others (public)	+	give access	x	execute
a	all				

`crypt [` *key* `]` *filename*
encrypt a file using *key* as the encryption key. To edit an encrypted file, use `vi -x`.

`ln [ -s ]` *oldname newname*
make a link to *oldname* called *newname* . With `-s`, make a symbolic link.

`ls` *option*
List files and selected properties. *option* can be one or more of:

–a	list hidden files.
–l	long listing. Shows permissions, links, owner, modification time, and name.
–lg	groups. Shows group ownership in addition to above properties.
–ld	directory. Shows `-l` listing for a directory itself, rather than the files it contains.
–F	Append a tag indicating the file type:

*	execute permission is set.
/	directory.
@	symbolic link.

`pushd`, `popd` and `dirs`
use the directory stack to remember and revisit directories.

`touch` *filename*
change a file's modification time to the current time. Create a file if *filename* doesn't exist.

`tty`
display the filename of the terminal.

`umask` *ddd*
set initial permissions mask for new files according to the table below. The default mask is `022`.

Files		*Directories*	
value	*permissions*	*value*	*permissions*
0	rw-	0	rwx
1	rw-	1	rw-
2	r--	2	r-x
3	r--	3	r--
4	-w-	4	-wx
5	-w-	5	-w-
6	---	6	--x
7	---	7	---

I/O Redirection

>	redirect the standard output.
>!	force redirection, even if the file exists.
>>	append the standard ouput to the file.
>>!	append the standard output, creating the file if necessary.
>&	redirect both the standard output and the standard error.
>>&	append both the standard output and the standard error.
<	redirect the standard input.

`|` pipe. Use the standard output of the command on the left as the standard input for the command on the right.

`|&` Use both the standard output and standard error of the command on the left as input for the command on the right.

`/dev/null`
the system wastebasket. Unwanted output can be redirected to this file.

`/dev/tty`
the terminal. Output from commands in scripts and subshells can redirected to the screen using this filename.

`set noclobber`
This command prevents files from accidental overwrites. Include it in your `.cshrc` file.

`| tee` *filename*
When placed on the end of a pipeline, the standard output is both redirected to *filename* and echoed on the screen.

Command-Line Special Characters

`&` run the command in the background.

`\`*c* escape character. Interpret *c* as text with no special meaning.

`"` double-quote. Interpret characters enclosed by double-quotes as a single word.

`'` quote. Interpret characters enclosed by quotes as a single word, and do not perform substitutions. (Special characters must still be escaped to be ignored.)

`;` command separator. Commands separated by semicolons are performed sequentially.

Filters

`cat` *filename ...*
: concatenate and print one or several files.

`fmt` *filename*
: simple file formatter.

`grep` *"reg_exp" filename ...*
: search for a regular expression in a file or files. *reg_exp* is a combination of text, escaped characters, and *grep* special characters from the following table:

character	*matches:*
`^`	The beginning of a text line.
`$`	The end of a text line.
`.`	Any single character (like ? in filename substitution).
`[...]`	Any single character in the bracketed list or range.
`[^...]`	Any character not in the list or range.
`*`	Zero or more occurrences of the *preceding character* or *regular expression*. (Not like filename substitution.)
`.*`	Zero or more occurrences of any single character. Equivalent to '`*`' in filename substitution.
`\`	Escapes special meaning of next character.

`head` [*-n*] *filename*
: Display the first *n* lines of a file.

`look` *str*
: look up words beginning with *str* in the system dictionary.

`more`
: page through a file. The subcommand:

/string	skips to a screenful containing *string*.

`nroff` *-mac filename*
: format a file using the *mac* macro package.

`pr -t` *-n filename*
: print a file in *n* column format. the `-t` option suppresses a heading that would otherwise appear.

`rev` *filename*
: reverse the order of characters in each line of a file.

`spell` *filename*
: check for misspelled words.

`sort` *filename*
: put lines of a file in order.

`tail` *option filename*
: display the last several lines of a file, as determined by *option*:

-*n* display the last *n* lines.

+*n* skip to line number *n*, and display the remaining lines.

`wc` *filename*
display the number of lines, words and characters in a file.

Job Control

`%` [*n*]
bring job *n*, or the current job, to the foreground.

`%` [*n*] `&`
resume processing stopped job *n*, or the current job, in the background.

`jobs`
display the list of background jobs.

Process Control

`kill` *PID*
terminate process number *PID*.

`ps [ -au ]`
display the list of processes. With the `-au` option, display the list of processes owned by all users.

User Activity

`grep` *userid* `/etc/passwd`
search for *userid* in file containing the list of local users.

`su` [*userid*]
switch userid to *userid*, or `root` (the superuser), when *userid* is omitted.

`w` display a detailed list of users currently logged in.

`who`
display a brief list of users currently logged in.

`who am i`
display the *userid*, terminal name, date and time.

`whoami`
display the userid only.

Managing Files

`diff` *leftfile rightfile*
show differences between two files.

`df` show disk space utilization on each disk as a percentage of capacity.

`du` show disk space utilization in the current directory.

`find`

`find` *pathname options*
locate files that meet the conditions specified in *options*, within the directory *pathname*, and its subdirectories. *options* can be:

`\!`*option*	invert the meaning of *option*. (Select files for which the option doesn't apply.)
`\(` *option* ... `\)`	group a set of options into one condition.

`-exec` *command* `'{}' \;`	perform *command* on the located files.
`-group` *group*	locate files belonging to *group*.
`-mtime` *n*	select files modified within *n* days.
`-name` *filename*	locate files that match *filename* after filename substitution.
`-newer` *checkfile*	locate files modified more recently than *checkfile*.
`-o`	within an option group of the form: `\(` *-option* `-o` *option* `\)` select files for which either option applies. Normally, a file is selected only when all options apply.
`-print`	print the list of selected files.
`-user` *userid*	select files owned by *userid*.
`file` *filename*	determine the type of device, or type of data contained in, *filename*.

`make`

`make [ -n ][ -f` *makefile* `]`

perform the procedure described in *makefile*. With the -n option, `make` echoes the commands it will perform, without performing them.

makefile is composed of *macro definitions* and *target definitions*:

macro definition

a line of the form:

macro = *expansion*

macro

is a character string.

expansion

is the remainder of the text on the line.

Once defined, macros are called as:

`$(`*macro*`)`

throughout the file.

target definitions

a set of lines of the form:

target: *dependency*...
    command line[51]

[51] starts with a TAB

. . .

target
a filename produced by, or logical label for, the step.

dependency
the name of another target upon which this one depends.

command line
a UNIX command line, beginning with a tab character. (If the tab is followed immediately by a dash (–) then return codes from commands on that line are ignored. Comment lines are introduced with a #).

`sccs`

`sccs` *subcommandfilename*
use a feature of the source code control system. *subcommand* is one of:

`create`	put a file under `sccs` control by creating a history file in the `SCCS` subdirectory.
`info`	report any files checked out (omit *filename* in this case).
`edit`	check out a file.
`diffs`	contrast the edited version with the most recent checked in version.
`delget`	check in a new version to the history file and replace the existing version of the text.
`delta`	check in a new version to the history file.
`get`	rebuild the current checked in version.
`prt`	examine the summary comments for all versions in the history file.

`sccsdiff` *–x.y* –r*m.n*
contrast previous verions *x.y* and *m.n*.

`tar`

`tar` *option filename*
tape archive program. *option* is one of:

`-cvf` *drive*	create an archive on *drive*.
`–xvf` *drive*	extract files from an archive on *drive*.
`–tvf` *drive*	display the files in the archive on *drive*.

Locating Commands

`whatis` *command*
give one-line description of a command.

`whereis` *command*
search the standard directories for the pathname of a command.

`which` *command*
search directories in the user's `path` variable for *command*.

Line Printer Commands

`lpr` [–P*printer*] *filename*
select a *printer* to print a file.

`lpq` [–P*printer*] *filename*
display the queue for *printer*.

`lprm` [–P*printer*] *job*
remove *job* from the queue for *printer*.

`troff -t` *options filename ...* > *output.file*
place typesetter-formatted output in an intermediate (binary) *output.file* for later printing.

`lpr -t` *output.file*
print a `troff` output file.

`screendump | rastrepl | lpr -v`
print the workstation screen display.

Misc. Commands

`chesstool`
window-based chess-playing program.

`csh`
the C-Shell command.

`date`
display the date and time.

`echo`
display the arguments on the terminal.

`printenv`
display the list of environment variables and values.

`set` *var* [=*value*]
create, or assign a value to, a C-Shell variable.

`sh` the Bourne shell command.

`source` *filename*
read and perform commands in *filename*.

`time` *command*
report statistics for *command*.

Index

Revision History

Version	Date	Comments
A	3 January 86	First edition of this Manual.

Notes

Notes

Notes

Notes

Notes

Doing More With UNIX: Quick Reference

This quick reference lists commands presented in this manual, including a syntax diagram and brief description.

1. Files

1.1. Filename Substitution

Wild Cards[1]	`? *`
Character Class	`[`*c...*`]`
Range	`[`*c–c*`]`

c is any single character.

String Class	`{`*str*`[,`*str*`]}`

str is a combination of characters, wild cards, embedded character classes and embedded string classes.

Home Directory	`~`
Home Directory of Another User	`~`*user*
List Hidden Files	`ls -[l]a`

1.2. File Properties

Seeing Permissions	`ls -l` *filename*
Changing Permissions	`chmod` *nnn filename* `chmod` *c=p...*[,*c=p...*] *filename*

n, a digit from 0 to 7, sets the access level for the user (owner), group, and others (public), respectively. *c* is one of: **u** – user, **g** – group, **o** – others, or **a** – all. *p* is one of: **r** – read access, **w** – write access, or **x** – execute access.

Setting Default Permissions	`umask` *ugo*

ugo is a (3-digit) number. Each digit restricts the default permissions for the user, group and others, respectively.

[1] from *Getting Started With UNIX*

Changing Modification Time	`touch` *filename*
Making Links	`ln` *oldname newname* `ln -s` *oldname newname*
Seeing File Types	`ls -F`

1.3. Encrypting Files

Source Files	`crypt <` *source* `>` *encrypted*
Editing	`vi -x` *encrypted*
Decrypting Files	`crypt <` *encrypted* `| more` `crypt <` *encrypted* `>` *text*

`crypt` asks for the encryption key.

1.4. Searching with `more`

Run `more`	`more` *filename*
Next Line[1]	RETURN
Next 11 Lines[1]	`d`
Next Page[1]	SPACE
Search for Pattern	`/`*pattern*
Next Occurrence	`n`
Next File	`:n`

1.5. The Directory Stack[2]

Change Directory, Push	`pushd` *directory*
Change to Top Directory, Pop	`popd`
Show Stack	`dirs`

2. Commands

2.1. Command-Line Special Characters

Quotes and Escape

Join Words	`"..."`
Suppress Filename, Variable Substitutions	`'...'`
Escape Character	`\`

[2] a feature of the C-Shell.

Separation, Continuation

Command Separation	`;`
Command-Line Continuation	`\`RETURN

2.2. I/O Redirection and Pipes

Standard Output	`>`
	 `>!`

Appending to Standard Output	`>>`
	 `>>!`

Standard Input	`<`
Standard Error and Output	`>&`
Standard Error Separately	`(` *command* `>` *output* `) >&` *errorfile*
Pipes/Pipelines	*command* `|` *filter* [`|` *filter*] ...
Duplicating Displayed Output	*command* `| tee` *filename*

Filters

Word/Line Count	`wc [-l]`
First *n* Lines	`head [-`*n*`]`
Last *n* Lines	`tail [-`*n*`]`
Skip to Line *n*	`tail [+`*n*`]`
Show Nonprinting Characters	`cat -v`
Sort lines	`sort [-`*n*`]`
Format Paragraphs	`fmt`
Reverse Character Order	`rev`
Multicolumn Output	`pr -t`
List Spelling Errors	`spell`
Substitutions in Output Stream	`sed -e "s/`*pattern*`/`*string*`/[g]"`
Report-Generation	`awk`[3]

2.3. Searching with `grep`

`grep` Command	`grep "`*pattern*`"` *filename* *command* `| grep "`*pattern*`"`

[3] see *Using UNIX Text Utilities*

`grep` Search Patterns

beginning of line	`^`
end of line	`$`
any single character	`.`
single character in list or range	`[...]`
character not in list or range	`[^...]`
zero or more of preceding character or pattern	`*`
zero or more of any character	`.*`
escapes special meaning	`\`

3. C-Shell Features

3.1. History Substitution

The History List

Set Up History List	**`set history=`***n*
See History List	**`history`** `[-h]`

Event Designators

Repeat Previous Command	`!!`
Display Previous Command	`!!:p`
Command Line *n*	`!`*n*
n Commands Back	`!-`*n*
Command Beginning with *str*	`!`*str*
Command Containing *str*	`!?`*str*`[?]`
All Arguments to Prev. Command	`!*`
Last Argument to Prev. Command	`!$`
First Argument to Prev. Command	`!^`
n'th Argument	`!:`*n*

Word Designators

All Arguments	`:*`
Last Argument	`:$`
First Argument	`:^`
n'th Argument	`:n`
Arguments *x* Through *y*	`:`*x*`-`*y*

Modifiers

Print Command Line	`:p`
Substitute Command Line	`:[g]s/`*l*`/`*r*`/`

3.2. Aliases

`alias` Command	**`alias`** *name 'definition'*

definition can contain escaped history substitution event and word designators as placeholders for command-line arguments.

3.3. Variable Substitution

Creating a Variable	**`set`** *var*
Assigning a Value	**`set`** *var* `=` *value*
Expressing a Value	`$`*var*
Displaying a Value	**`echo`** `$`*var*

value is a single word, an expression in quotes, or an expression that results in a single word after variable, filename and command substitution takes place.

Assigning a List	**`set`** *var* `=` *(list)*

list is a space-separated list of words, or an expression that results in a space-separated list.

Selecting the *n*'th Item	`$`*var*`[`*n*`]`
Selecting all Items `$`*var*	
Selecting a Range	`$`*var*`[`*x*`-`*y*`]`
Item Count	`$#`*var*

3.4. `foreach` Lists

Start `foreach` Loop	**`foreach`** *var (list)*

`foreach` prompts for commands to repeat for each item in *list* (with >), until you type **end**. Within the loop, `$`*var* stands for the current item in *list*.

3.5. Command Substitution

Replace Command with its Output on the Command Line	`` `...` ``

3.6. Job Control

Run Command in the Background	`&`
Stop Foreground Job	CTRL-Z
List of Background Jobs	**`jobs`**
Bring Job Forward	`%[`*n*`]`
Resume Job in Background	`%[`*n*`]`

4. Processes

Listing	**`ps`** `-[aux]`
Terminating	**`kill`** `[-9]` *PID*
Timing	**`time`** *command*
Scheduling	**`at`** *time*[a|p] *script*

time is a number up to 4 digits. *script* is the name of a file containing the command line(s) to perform.

5. Users

Seeing Who Is Logged In	**`who`** **`w`**
Changing Identities	**`su`** *[username]*

Seeing Your User Name	**`whoami`**
	`who am i`
	`who is this`

6. Managing Files

6.1. Looking Up Files

Standard Commands	**`whereis`** *file*
Aliases and Commands	**`which`** *command*
Describe Command	**`whatis`** *filename*
Searching Out Files	

`find` *dir* **`-name`** *name* **`-print`**

dir is a directory name within which to search. *name* is a filename to search for.

6.2. Tracking Changes

Comparing Files	`diff` *leftfile rightfile*

`diff` prefixes a left angle-bracket (<) to selected lines from *leftfile* and a right angle bracket (>) to lines from *rightfile*.

Auditing Changes

Putting Files Under `sccs`	`mkdir SCCS`
	`chmod 775 SCCS`
	`sccs create` *filename* ...
	`rm ,*`

Checking Files Out	`sccs edit` *filename* ...
Checking Files In	`sccs delget` *filename* ...
Backing Files Out	`sccs unedit` *filename* ...
Recovering Current Versions	`sccs get SCCS`
Reviewing Pending Changes	`sccs diffs` *filename* ...

6.3. Automating Tasks

Create a Makefile	`vi Makefile`

A makefile consists of macro definitions and targets.

Test Makefile	`make -n` [*target*]
Run `make`	`make` [*target*]

6.4. Managing Disk Usage

Seeing Disk Usage	`df`
	`du -s`
	`du | sort -r -n`
	`ls -l`

Making A Tape Archive	`tar -cv[f` *drive*] *file* ...
Extracting Archived Files	`tar -xv[f` *drive*] *file* ...

[3] See Chapter 6 for details.

7. Printing

7.1. The Printer Queue

List the Queue	`lpq`
Removing a Printer Job	`lprm` *job*
Removing Your Printer Jobs	`lprm -`
Selecting a Printer	`lpr -P`*printer* `lpq -P`*printer* `lprm -P`*printer job*

7.2. Printing `troff` Output and Screen Dumps

`troff` Output	`lpr -t`
Screen Dumps	`screendump [| rastrepl] | lpr -v`

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 287815

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Berkshire House
High Street
Ascot, Berkshire SL5 7HU
England 0990 28911
TLX 846573

Germany: 089 416-00820
UK: 0990 25942
France: 01 630 23 24

Canadian Headquarters
416 477-6745

Europe, Middle East, and Africa, call European Headquarters:
0990 28911

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales