



A/UX® User Interface

🍏 APPLE COMPUTER, INC.

© 1990, Apple Computer, Inc., and UniSoft Corporation. All rights reserved.

Portions of this document have been previously copyrighted by AT&T Information Systems and the Regents of the University of California, and are reproduced with permission. Under the copyright laws, this manual may not be copied, in whole or part, without the written consent of Apple or UniSoft. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. Under the law, copying includes translating into another language or format.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

Apple, the Apple logo, A/UX, ImageWriter, LaserWriter, and Macintosh are registered trademarks of Apple Computer, Inc.

MacPaint is a registered trademark of Claris Corporation.

UNIX is a registered trademark of AT&T Information Systems.

Simultaneously published in the United States and Canada.

**LIMITED WARRANTY ON MEDIA
AND REPLACEMENT**

If you discover physical defects in the manual or in the media on which a software product is distributed, Apple will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged software media and manuals for as long as the software product is included in Apple's Media Exchange Program. While not an upgrade or update method, this program offers additional protection for up to two years or more from the date of your original purchase. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different, check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

A/UX User Interface

Contents

Preface

Chapter 1 The A/UX Shells: An Overview

Chapter 2 Bourne Shell Reference

Chapter 3 Korn Shell Reference

Chapter 4 C Shell Reference

Chapter 5 Shell Layering

Appendix A Additional Reading

Preface

Conventions Used in This Manual

Throughout the A/UX manuals, words that must be typed exactly as shown or that would actually appear on the screen are in `Courier` type. Words that you must replace with actual values appear in *italics* (for example, *user-name* might have an actual value of `joe`). Key names appear in CAPS (for example, RETURN). Special terms are in **bold** type when they are introduced; many of these terms are also defined in the glossary in the *A/UX System Overview*.

Syntax notation

All A/UX manuals use the following conventions to represent command syntax. A typical A/UX command has the form

```
command [flag-option] [argument] ...
```

where:

- | | |
|----------------------|---|
| <code>command</code> | Command name (the name of an executable file). |
| <i>flag-option</i> | One or more flag options. Historically, flag options have the form

<code>-[opt...]</code>

where <i>opt</i> is a letter representing an option. The form of flag options varies from program to program. Note that with respect to flag options, the notation

<code>[-a][-b][-c]</code>

means you can select one or more letters from the list enclosed in brackets. If you select more than one letter you use only one hyphen, for example, <code>-ab</code> . |
| <i>argument</i> | Represents an argument to the command, in this context usually a filename or symbols representing one or more filenames. |
| <code>[]</code> | Surround an optional item. |

- ... Follows an argument that may be repeated any number of times.
- Courier type anywhere in the syntax diagram indicates that characters must be typed literally as shown.
- italics* for an argument name indicates that a value must be supplied for that argument.
- Other conventions used in this manual are:
- <CR> indicates that the RETURN key must be pressed.
- ^*x* An abbreviation for CONTROL-*x*, where *x* may be any key.
- cmd(sect)* A cross-reference to an A/UX reference manual. *cmd* is the name of a command, program, or other facility, and *sect* is the section number where the entry resides. For example, `cat(1)`.

Chapter 1

The A/UX Shells: An Overview

Contents

1. Using the shell	1
1.1 How the shell interprets and manages commands	1
1.2 Programming constructs	2
2. The shell environment	2
3. When the shell executes commands	4
3.1 Standard input and output	6
4. The three A/UX shells	9
4.1 The Bourne shell	9
4.2 The Korn shell	9
4.3 The C shell	9
4.4 Similarities among the shells	10
4.4.1 Command features	10
4.4.2 Shell layering and job control	10
4.4.3 The environment	11
4.4.4 Shell programming	11
4.5 Basic differences among the shells	11

Figures

Figure 1-1. The environment and new shell instances	4
Figure 1-2. How the shell executes a process	6
Figure 1-3. Input and output redirection	8

Tables

Table 1-1. Command features	12
Table 1-2. The environment	12
Table 1-3. Shell programming	13

Chapter 1

The A/UX Shells: An Overview

1. Using the shell

The A/UX shells are interactive programs that accept and interpret command input. The shells include standard features that simplify access to A/UX programs and allow you to combine programs, manipulate their input and output, run more than one program at a time, switch back and forth between jobs, and so on.

When you log in, the system invokes a shell (called the **login shell**) for you to work with. By default, this is the Bourne shell (`sh`). You can change your default shell to either the C shell (`csh`) or the Korn shell (`ksh`) by using the `chsh` command (see `chsh(1)` in *A/UX Command Reference*) or directly modifying a field in the system file `/etc/passwd`. See *A/UX Local System Administration* for more information.

1.1 How the shell interprets and manages commands

The shell prints a **prompt** character and waits (indefinitely) for you to give commands. In the Bourne shell, the prompt character is a dollar sign (\$) by default.

When you enter a command, the shell interprets it, looking for filenames, variables, and special characters (called **metacharacters**) that tell the shell what to do. Once the shell has interpreted all the information associated with your command, the shell organizes the information, sends it to the appropriate A/UX program (or starts the appropriate process), and waits for that program or process to execute.

You can instruct the shell to return the shell prompt immediately, without waiting for command completion, by putting your command in the “background.”

The shell presents a consistent interface: most commands can run in the foreground or background, be interrupted, take input from a file and send output to a file, and so on for any of the shell features for handling commands. The shell responds the same way whether you invoke any A/UX command, a built-in shell command (each shell has its own set of built-in commands), or a **shell program** (a program you can write using A/UX commands and built-in shell commands).

1.2 Programming constructs

The shell contains built-in commands and constructs that allow it to function as a programming language. You can write shell programs (often called **shell scripts**) that contain lists of commands for tasks that are complicated or that you have to repeat many times. Within shell programs you can

- refer to files in the current directory or another directory
- accept input from the keyboard or from a file
- direct output to the terminal screen or to a file
- loop, compare, and make decisions
- store information in variables
- invoke other programs (binary executables) and shell scripts

Shell programs may be typed in at the shell prompt or inserted in text files. See “Overview of Shell Programming” in the chapter of this manual that is appropriate for your shell.

2. The shell environment

The **environment** is a list of characteristics describing a particular user. These characteristics identify the user to the system, and influence and constrain that user’s access to the system. A user’s environment includes several attributes, including the search path, the user ID, the user’s file permissions, the current working directory, and a number of other environment variables. Some of these attributes can be modified by the user. See “The Environment” in the appropriate chapter of this manual.

If you assign new values in the environment, you can customize certain aspects of how the shell responds. You can also define and assign your

own variables; for example,

```
a=10
```

You have a choice about whether to keep these variables and their new values local, or insert them in the environment. In the Bourne shell, you insert variables and values in the environment using the `export` command; for example,

```
export a
```

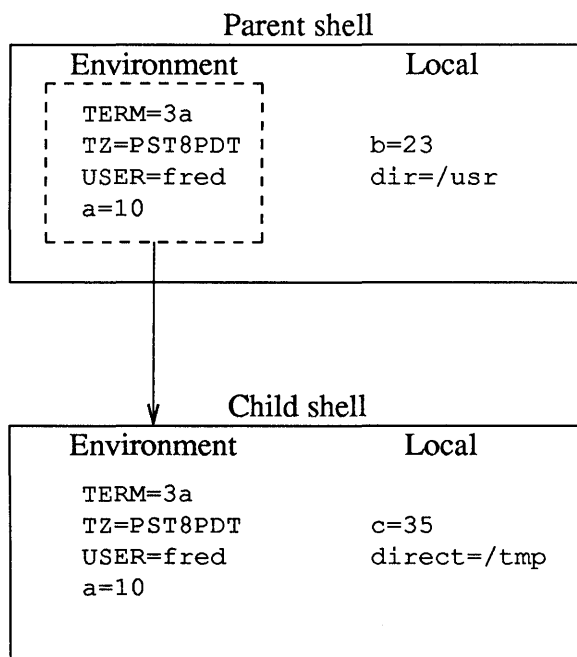
When you invoke a new instance of the shell, for example, using the command

```
sh
```

all the variables and values in the environment are *copied* for the new instance of the shell. Local variables are not copied and are not available in the new instance of the shell.

Figure 1-1 shows that the environment of a new instance of the shell contains all the values in the parent shell's environment. Note that local values, which were not inserted in the parent shell's environment, are not copied to new instances of the shell.

Figure 1-1. The environment and new shell instances



Thus, if you are working in the shell and want to create a new copy, or instance, of the shell, the new copy inherits all the values in your environment. This new instance is called a **subshell** or “child” shell. The original shell is called the “parent” shell. See “The Environment” and “The Environment and New Shell Instances” in the appropriate chapter of this manual, and “Commonly Used Environment Variables” in Chapters 2 and 3 and “C Shell Variables” in Chapter 4.

3. When the shell executes commands

To understand what occurs when you enter a command at the shell prompt, you should know the following:

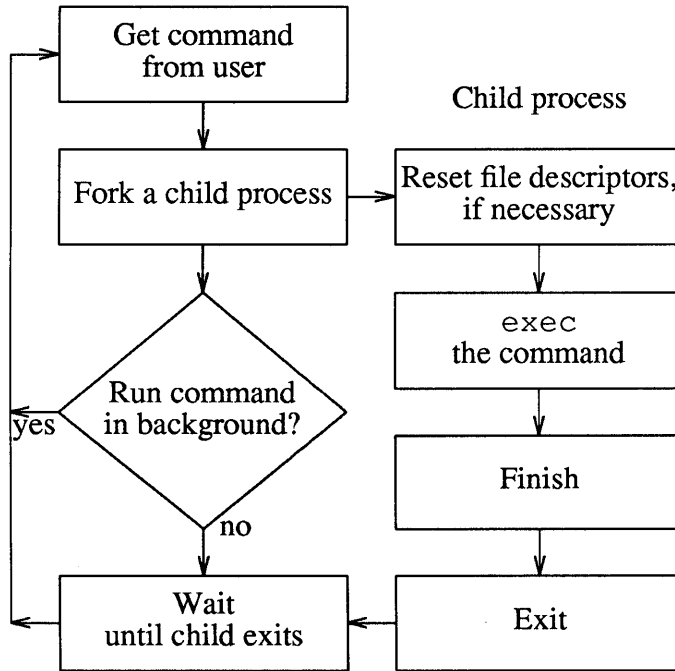
fork When a process “forks,” it creates another process like itself. There are a few differences between the original (or

parent) process and the new one (child). One of the differences is that the parent knows that it has a child running and receives a signal when the child terminates. `fork` can be thought of as the initialization phase of creating a new process.

- `exec` When a process “execs,” it overwrites the old process with a new one. That is, the original process is replaced by a new one. `exec` can be thought of as the program load phase of creating a new process.
- `wait` When a process “waits,” it simply waits for a change in the child’s status (usually termination) before resuming.

Figure 1-2 shows how the shell executes a process.

Figure 1-2. How the shell executes a process



See the next section for information about the file descriptors in Figure 1-2.

Note that the original process does not wait when you end your command line with an ampersand (&). In this case, the command executes as shown above, but the shell prints a new prompt so it can accept input immediately.

3.1 Standard input and output

The shell has certain defaults for where it expects to find input to a command, where it writes the command's output, and where it prints the error messages. These defaults are as follows:

Standard input (the data stream used for input to a command): Unless you use the less-than sign (<) on your command line, the shell expects any input to be typed in at the keyboard. The less-than sign (<) directs the shell to accept input from a file or device.

Standard output (the data stream used for output from a command): Unless you use the greater-than sign (>) on your command line, the shell directs any output to the terminal. The greater-than sign (>) directs the shell to write output to a file or device.

Standard error output (the data stream used for error messages from a command): Unless redirected, the shell directs standard error output to the terminal. Each shell uses its own method of redirecting standard error output; this is described under “I/O Redirection” in the appropriate chapter in this manual.

The possibilities for redirecting standard input, standard output, and standard error output are shown in Figure 1-3. In this figure, the `cat` command is the representative “process.” The same principles of redirection apply to any process.

Note that the `cat` command and many other A/UX commands can accept input from a file without using the less-than sign (<) by simply specifying the filename on the command line; that is,

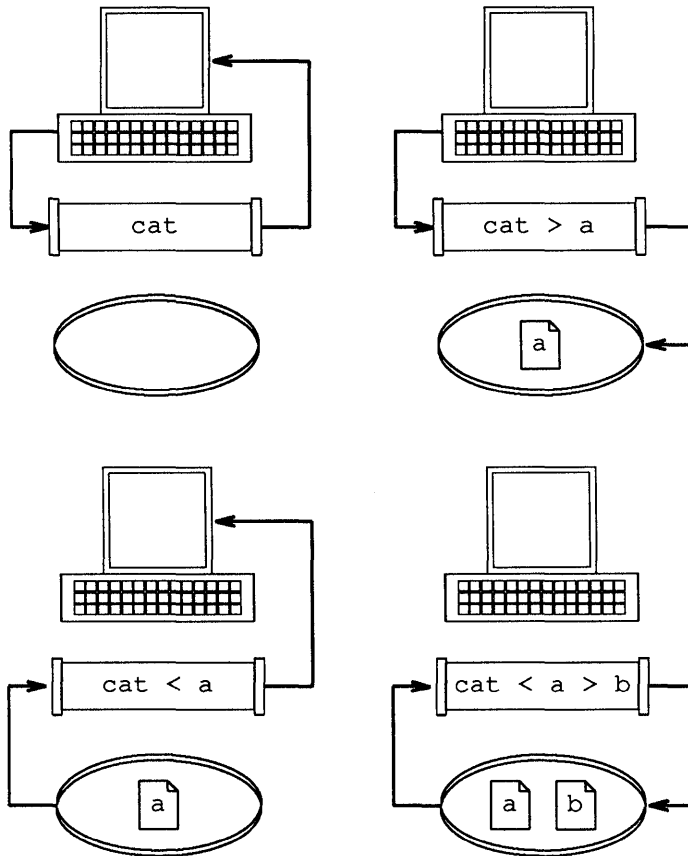
```
cat filename
```

has the same effect as

```
cat < filename
```

The symbol is shown in Figure 1-3 to illustrate input and output redirection.

Figure 1-3. Input and output redirection



See “Using Shell Metacharacters” and “I/O Redirection” in the appropriate chapter of this manual.

4. The three A/UX shells

The A/UX system provides three separate shell programs: the Bourne shell, Korn shell, and C shell. The Bourne shell is the default login shell if no other shell has been specified. See `chsh(1)` in *A/UX Command Reference* for information about changing your default login shell.

While each of these shells is a slightly different program, their basic function—interpreting commands for execution—is the same. They differ only in certain features that are built in. In general, if you know how to use one shell, you will not have much trouble figuring out how to use the others.

4.1 The Bourne shell

The Bourne shell is efficient and fast, and provides extensive constructs for shell programming. Shell programs written in the Bourne shell generally run two to five times faster than scripts in the C shell, and are easier to debug and get running. However, people familiar with the C programming language may find the Bourne shell's constructs less convenient than those of the C shell.

In general, the Bourne shell makes fewer concessions to interactive use than the Korn or C shells.

4.2 The Korn shell

The Korn shell is an extension of the Bourne shell, so it is compatible in many ways. It retains and extends the Bourne shell's capabilities and efficiency in programming, while also incorporating features that make giving commands, editing commands, and reusing commands much easier.

The Korn shell also provides job control: this allows you to switch back and forth between different jobs, and to begin working on a job and then put it in the background where it will continue (or wait) until you bring it back into the foreground.

4.3 The C shell

The C shell is not strictly compatible with the Bourne and Korn shell. It lacks many of the programming capabilities of the Bourne and Korn shells (including file descriptor assignment and most error handling), and it can use variables that are not compatible with the Bourne shell or Korn shell, such as the C shell variable that protects the user from

inadvertently overwriting files.

The C shell's programming constructs look much like the C programming language. In general, scripts written in the C shell execute more slowly, and some people find that they are more difficult to debug than scripts in the other shells.

Like the Korn shell, the C shell provides command history, editing, and reuse, as well as job control. Job control allows you to switch back and forth between different jobs, and to begin working on a job and then put it in the background where it will continue (or wait) until you bring it back into the foreground.

4.4 Similarities among the shells

Features that are common to all three A/UX shells are listed below.

4.4.1 Command features

All three shells provide the following features that are useful in entering commands at the prompt:

- background commands
- input and output redirection
- filename expansion with metacharacters
- pipelines
- multiple commands on a single line
- executing commands in a subshell

See “Interactive Use,” “Using Shell Metacharacters,” “Command Grouping,” and “Working With More Than One Shell” in the appropriate chapter of this manual.

4.4.2 Shell layering and job control

The `shl` program allows you to create up to seven labeled subshells called **shell layers** within your login shell. These layers can then be referred to by name (or number), suspended and resumed, deleted, and so on. Each of these layers appears like your login shell, but can be used to run a process while you switch to another layer. This provides a management scheme for multiple concurrent processes.

In addition, the C shell and Korn shell provide a form of job control that allows you to run multiple processes that can be suspended and resumed as desired.

See “Using Shell Layering,” “Working With More Than One Shell,” and “The Environment and New Shell Instances” in the appropriate chapter of this manual. Also see “Job Control” in the Korn and C shell chapters.

4.4.3 The environment

All three shells provide the following features in their environment:

- variables used during shell execution
- file(s) that may assign values to variables

See “The Environment,” “Environment Variables,” and “The `.profile` File,” “The `.kshrc` File,” or “The `.cshrc` File” in the appropriate chapter of this manual.

4.4.4 Shell programming

All three shells provide the following programming capabilities:

- examine exit status of jobs
- assign, reassign, and remove variables
- check file status
- compare and make decisions
- loop
- read and write information to and from standard input and output streams

See “Overview of Shell Programming,” “Exit Status: The Value of a Command,” “Assigning Values,” “Arithmetic Expressions,” “File Status,” “Control-Flow Constructs,” and “Input and Output” in the appropriate chapter of this manual.

4.5 Basic differences among the shells

Tables 1-1 through 1-4 summarize the basic differences in the capabilities of the three A/UX shells. In general, only relatively major differences have been included. Features such as different options to the same command, or different commands performing the same

function, are not mentioned here.

Note: Different entries in each column of the tables mean that the shells differ significantly at the level of capability, not in syntax or command name.

Table 1-1. Command features

Capability	sh	ksh	csH
command history	n	y	y
command reuse	n	y	y
command editing	n	y	y
tilde substitution	n	y	y
preceding directory remembered	n	y	n
exclude characters [!...]	y	y	n
repeated substitution in filename expansion	n	n	y
I/O redirection safety	n	n	y
connect background pipe to parent shell	n	y	n
job control	n	y	y

Note: Table 1-2 does not include information about differences in environment variables and shell execution options.

Table 1-2. The environment

Capability	sh	ksh	csH
.cshrc/.kshrc read at each invocation	n	y	y
.profile read at login	y	y	n
.login read at login	n	n	y
.logout read at logout	n	n	y
exportable functions	n	y	n
functions that may not be exported	y	n	n
exportable command aliases	n	y	n
command aliases that may not be exported	n	n	y

The `.kshrc` file is read at every invocation of the Korn shell if and only if the `ENV` variable has been set. See “The `.kshrc` File” in Chapter 3.

Table 1-3. Shell programming

Capability	sh	ksh	csH
functions	y	y	n
functions with local variables	n	y	n
typed variables	n	y	n
one-dimensional arrays	n	y	n
one-dimensional arrays of strings	n	y	y
creation of substrings	n	y	n
modifiers in substitution	n	n	y
substitution of defaults	y	y	n
built-in formatted output	y	y	n
built-in fixed output	n	n	y
file descriptors in redirection	y	y	n
redirection with control structures	n	y	n
built-in arithmetic	n	y	y
background exit status	y	y	n

Chapter 2

Bourne Shell Reference

Contents

1. The Bourne shell prompt	1
1.1 The secondary shell prompt	1
1.2 Changing the prompt character	1
2. Types of commands	1
3. The parts of a command	2
4. Interactive use	3
4.1 Command termination character	3
4.2 Impossible commands	3
4.3 Background commands	4
4.3.1 Checking command status	4
4.3.2 Logging out	4
4.4 Canceling commands	5
4.4.1 Before you press RETURN	5
4.4.2 While a command is running	5
4.4.3 Canceling background commands	7
5. Using shell metacharacters	7
5.1 Specifying filenames with metacharacters	9
5.2 Input and output redirection	11
5.3 Combining commands: pipelines	12
5.4 Command grouping	13
5.5 Conditional execution	15
5.6 Quoting	15
6. Working with more than one shell	17
6.1 Changing to a new shell	18
6.2 Changing your default shell	18
7. The environment	18
7.1 Listing existing values	19

7.2	Assigning values to environment variables	19
7.3	Removing environment variables	20
7.4	Commonly used environment variables	20
7.5	The environment and new shell instances	22
7.6	Special environments	22
7.7	The default environment on your system	24
8.	The <code>.profile</code> file	25
8.1	A sample <code>.profile</code> file	25
8.1.1	Locating commands	25
8.1.2	Shortcuts in changing directories	26
8.1.3	Receiving mail	27
8.1.4	Your editing environment	27
8.2	Customizing your login procedure	27
9.	Shell execution options	28
9.1	Options that affect the environment	28
9.2	Options for invoking new shells	29
10.	Restricted shell	29
11.	Using shell layering	30
12.	Overview of shell programming	30
12.1	Writing shell programs	31
12.2	Executing shell scripts	31
12.3	Comments	33
12.4	Writing interactive shell scripts	33
12.5	Canceling a shell script	33
12.6	Writing efficient shell scripts	34
13.	Command evaluation	34
13.1	Forcing more than one pass of evaluation	36
13.2	Command execution	36
13.3	Exit status: the value of the command	37
14.	Defining functions	37
15.	Positional parameters and shell variables	39
15.1	Positional parameters	39
15.1.1	Setting values in a script	40
15.1.2	Changing parameter positions	41
15.1.3	Number of parameters	42

15.2 Shell variables	42
15.2.1 Assigning values	42
15.2.2 Removing shell variables	43
15.3 Setting constants	44
15.4 Parameter and variable substitution	44
15.5 Testing assignment and setting defaults	45
15.6 Parameters and variables set by the shell	46
16. Control-flow constructs	47
16.1 for loops	47
16.2 case statements	49
16.3 while loops	51
16.4 until loops	53
16.5 if then else	54
16.6 exit	56
17. Input and output	57
17.1 I/O redirection	57
17.1.1 Redirection with file descriptors	57
17.1.2 File descriptors redirecting input	58
17.1.3 File descriptors redirecting output	58
17.1.4 Combining standard error and standard output	58
17.1.5 Changing the shell's standard input and output	59
17.1.6 Associating file descriptors with other files	60
17.2 Reading input	61
17.3 Taking input from scripts	62
17.4 Using command substitution	65
17.5 Writing to standard output	66
18. Other features	68
18.1 Arithmetic and expressions	68
18.2 File status and string comparison	69
18.3 The null command (:)	70
19. Error handling	71
19.1 Fault handling and interrupts	71
19.2 Debugging a shell script	75

20. Summary of Bourne shell commands 75

Chapter 2

Bourne Shell Reference

1. The Bourne shell prompt

The Bourne shell is a program that interprets commands and arranges for their execution. The Bourne shell displays a character called the **prompt** (or **primary shell prompt**) whenever it is ready to begin reading a new command from the terminal. By default, the Bourne shell prompt character is set to the dollar sign (\$).

1.1 The secondary shell prompt

If you press the RETURN key when the shell expects further input, you will see the **secondary shell prompt**. By default, this prompt character is set to the greater-than sign (>). Like the primary shell prompt, this can be redefined.

The secondary prompt will appear, for example, if you enter a multiline construct (such as a function definition) at the primary shell prompt. The secondary prompt will appear at each line until you give the final delimiter. Whenever you have a secondary prompt (either because you are using a multiline construct or because of an error), an *interrupt* will abort the process and issue a primary prompt (\$) for another command. See “Canceling Commands” for information about the *interrupt* on your system.

1.2 Changing the prompt character

You may change the primary prompt character by redefining the environment variable PS1 to any other character or string of characters. You can change the secondary prompt character by redefining the PS2 environment variable. See “Commonly Used Environment Variables.”

2. Types of commands

The shell works with three types of commands:

Built-in shell commands: Built-in commands are written into the shell itself and are generally used for writing shell programs. Each

A/UX shell has a slightly different set of built-in commands. The built-in Bourne shell commands are listed under “Summary of Bourne Shell Commands.”

A/UX commands: Every shell can invoke all A/UX commands (see “Command Summary by Function” in *A/UX Command Reference* for a complete list of these). A/UX commands are executable programs stored in system directories such as `/bin` and `/usr/bin`. When you enter an A/UX command (for example, `ls`), the shell searches all directories specified by your `PATH` variable (see “Locating Commands”) to locate the program and invoke it.

User-defined commands: You can combine built-in shell commands and A/UX commands to define your own **shell programs** (see “Overview of Shell Programming”). Shell programs can be typed in at the shell prompt or entered in a file. A shell program contained in a file is generally called a **shell script**. Once a shell script is defined, with certain limitations, it can be used like any other command or program.

You can also write your own commands in a high-level language such as C. (See *A/UX Programming Languages and Tools, Volume 1* for more information.) The names of user-defined commands should not be the same as any existing shell or A/UX command.

3. The parts of a command

Whenever you see a shell prompt, you can enter a command by typing the command name. Most A/UX commands have one or more **flag options**, which can follow the command name to modify the way the command operates. Flag options are usually a hyphen followed by one or more characters; for example, `-l` is a flag option to the `ls` command:

```
ls -l
```

In this case, the `-l` is a flag option that modifies the way the `ls` command operates, producing a long listing that contains more information than the standard `ls` output. For the flag options that apply to a particular A/UX command, see the manual page entry for

that command in *A/UX Command Reference*. For options to the Bourne shell built-in commands, see “Summary of Bourne Shell Commands.”

Many A/UX commands also expect one or more **arguments**, which pass information to the command. An argument may be any data expected by the command; for example, a directory name may be an argument to the `ls` command:

```
ls /bin
```

The entire command specification, including any flag options and other arguments, is called the **command line**. A command line is terminated by RETURN. For example, in the command line

```
ls -l /bin
```

`ls` is the command name, `-l` is a flag option (specifying a long listing), and `/bin` is an argument (specifying which directory to list).

To give a command longer than one line, you must precede RETURN with a backslash (`\`). This prevents the shell from interpreting RETURN as the end of a command. You can continue this for several lines; the shell will wait for a plain RETURN (not preceded by a backslash) to execute the multiline command.

Commands can also be combined; see “Command Grouping.”

4. Interactive use

4.1 Command termination character

When you are entering commands to the shell interactively, the shell will not begin executing the command until you press the RETURN key. Therefore, if you mistype something, you can back up and correct the mistake before pressing RETURN. When the shell recognizes the RETURN, it executes the command line; when the process completes, a new prompt will be printed on the screen. The shell is now ready to accept further commands.

4.2 Impossible commands

If you give an impossible command (a command that doesn’t exist or a command line that uses improper syntax), the shell will print an error message and return the prompt for another command.

4.3 Background commands

You can direct the shell to execute commands in the “background” while you continue to work at the shell prompt (the “foreground”). To run background processes, end the command line with an ampersand (&) before the final RETURN. For example,

```
cat smallfile1 smallfile2 > bigfile &  
1234
```

The number shown below the command line is the **process ID (PID)** associated with the sample `cat` command as long as it is executing. After the process ID is displayed, the shell returns the prompt so you can use the terminal immediately for other work.

Note: To save the output from a job you are running in the background, you must redirect it into a file or pipe it to a printer. If you do not redirect the command output, it will appear on your screen and will not be saved. In addition, remember that the output of a background command is not complete until the command has finished. The presence of a prompt does not mean that the output is ready for use.

To suspend processes that require input from the keyboard (such as an editor or a remote login across a network), use shell layering. See Chapter 5, “Shell Layering.”

4.3.1 Checking command status

To check on the status of a background command, use

```
ps
```

This command shows the **process status** of all your commands; they are identified by process ID and by name. See `ps(1)` in *A/UX Command Reference* for details.

4.3.2 Logging out

The shell terminates all processes when you log out of the system (or are forced to log out, for example, by a broken dialup connection). To make sure that a process will continue to execute after you log out, use the `nohup` command (which stands for “no hang up”) as follows:

`nohup command &`

See `nohup(1)` in *A/UX Command Reference* for details.

4.4 Canceling commands

A number of special control sequences come into play when canceling commands. The A/UX standard distribution defines these sequences as follows:

Name	A/UX standard distribution
<i>interrupt</i>	CONTROL-c
<i>quit</i>	CONTROL-l
<i>erase</i>	DELETE
<i>kill</i>	CONTROL-u
<i>eof</i>	CONTROL-d
<i>swtch</i>	CONTROL-'
<i>susp</i>	CONTROL-z

However, you may reassign any of these sequences using the `stty` command. See `stty(1)` in *A/UX Command Reference* for more information.

4.4.1 Before you press RETURN

If you type part of a command and then decide you do not want to execute it, you can send an *interrupt* or *kill* to the system at any point in the command line.

4.4.2 While a command is running

There are several ways to stop a command that is executing. You can redefine these using `stty` unless otherwise noted.

- Send the *interrupt* signal.

For example, the output of a command such as

```
cat /etc/termcap
```

will scroll by on your terminal. If you want to terminate the process, you can send the *interrupt* signal. Because the `cat` command does not take any precautions to avoid or otherwise handle this signal, the *interrupt* will (eventually) cause it to terminate.

- Use `CONTROL-s` to suspend scrolling output.

The A/UX control-flow keys are `CONTROL-s` (suspend scrolling output) and `CONTROL-q` (resume scrolling output). You can use these to stop a screenful of output, resume scrolling, and stop a screenful again. `CONTROL-s` and `CONTROL-q` cannot be redefined using `stty`; however, `stty` can enable and disable control-flow using “`stty -ixon`”.

- Send an *eof* character.

Many programs (including the shell) terminate when they get an *eof* condition from their standard input. You could accidentally terminate the shell (which would log you off the system) if you enter *eof* at a prompt or, in terminating some other program, if you send an *eof* one time too many.

- Wait for the *eof* condition from a file.

If a command has its standard input redirected from a file, then it will terminate normally when it reaches the end of that file. If you give the command

```
mail ellen < note
```

(where `note` is an existing file), the `mail` program will terminate when it detects the *eof* condition from the file.

- Send the *quit* signal.

If you run programs that are not fully debugged, it may be necessary to stop them abruptly. You can stop programs that hang or repeat inappropriately by using *quit*. This will usually produce a message such as

```
Quit (Core dumped)
```

indicating that a file named `core` has been created containing information about the state of the running program when it terminated because of the *quit* signal. You can examine this file yourself, or forward information to the person who maintains the program telling him or her where the `core` file is.

- Send a *suspend* signal.

If you are using shell layering, you can type `suspend` to stop jobs temporarily that are running on a shell layer. You can then resume the job with a special `shl` command. See Chapter 5, “Shell Layering.”

4.4.3 Cancelling background commands

If you have a job running in the background and decide you do not want the command to finish executing, use the A/UX `kill` command.

When a job is running in the background, it ignores *interrupt* and *break* signals. To terminate a background command, use

```
kill process-ID
```

The `kill` command takes the process ID as an argument. See `kill(1)` and `ps(1)` in *A/UX Command Reference* for details.

5. Using shell metacharacters

Shell **metacharacters** are characters that perform special functions in the shell. This section discusses how to use these metacharacters. The following are the Bourne shell metacharacters:

- & An ampersand at the end of a command line causes the shell to run the command(s) in the background and prints the process ID(s).
- ? A question mark used as part of a file or directory name causes the shell to match any single character (except a leading period).
- * An asterisk used as part of a file or directory name causes the shell to match zero or more characters (except a leading period).
- [] Brackets around a sequence of characters (except the period) cause the shell to match each character one at a time.
- A hyphen used within brackets to designate a range of characters (for example, [A-Z]) causes the shell to match each character in the range.
- < A less-than sign following a command and preceding a filename causes the shell to take the command’s input from that file.
- > A greater-than sign following a command and preceding a filename causes the shell to redirect the command’s standard

output into the file. See “Input and Output” for a description of how this metacharacter is used to redirect error output.

- >> Two greater-than signs following a command and preceding a filename cause the shell to append the command’s output to the end of an existing file.
- | A vertical bar between two commands on a command line causes the shell to redirect the output of the first command to the input of the second command. This can occur multiple times on a command line, forming a pipeline.
- ; A semicolon between two commands on a command line causes the shell to execute the commands sequentially in the order in which they appear.
- { } Braces around a series of commands group the output of the commands.
- () Parentheses around a pipeline or sequence of pipelines cause the whole series to be treated as a simple command (which may in turn be a component of a pipeline), and a subshell to be spawned for the commands’ execution.
- \ A backslash prevents the shell from interpreting the metacharacter that follows it.
- ' ' Single quotes around a command, a command name and argument, or an argument prevent the shell from interpreting the enclosed metacharacters.
- " " Double quotes around a command, an argument, or a command name and argument prevent the shell from interpreting the enclosed metacharacters with the exception of back quotes (` `) and the dollar sign (\$).
- ` ` Back quotes around a command cause the characters in that command to be replaced with the output from that command.
- \$ The dollar sign causes evaluation of the variable it precedes. “\$a” causes evaluation of the variable a.

5.1 Specifying filenames with metacharacters

Using the filename expansion metacharacters (also called “wildcards”) will spare you the job of typing long lists of filenames in commands, looking to see exactly how a filename is spelled, or specifying several filenames that differ only slightly.

These metacharacters are interpreted and take effect when the shell evaluates commands. At this point, the word incorporating the metacharacter(s) is replaced by an alphabetic list of filenames, if any are found that match the pattern given. Filename expansion metacharacters can be used in any type of command; however, in the case of filenames given for input and output redirection, filename expansion may cause unexpected results if the metacharacter usage expands into more than a single filename. To turn off the special meaning of metacharacters and use them as ordinary letters, they must be quoted. See “Quoting.”

The following are filename expansion metacharacters in the Bourne shell:

- ? A question mark matches any single character in a filename. For example, if you have files named

```
a bb ccc dddd
```

the command

```
echo ???
```

matches a sequence of any three characters and returns

```
ccc
```

- * An asterisk matches any sequence of characters, including the empty sequence, in a filename. (It will not, however, match the leading period in such names as `.profile`.) To list the sequence of files named

```
chap chap1 chap2 chap3 chap3A chap12
```

you can use the notation

```
ls chap*
```

The files are listed as

```
chap chap1 chap12 chap2 chap3 chap3A
```

Note that in the first file listed, `chap`, the asterisk matched the null sequence composed of no characters.

- [] Brackets enclosing a set of characters match any *single* character, one at a time, from the set of enclosed characters. Thus,

```
ls chap.[12]
```

matches the filenames

```
chap.1 chap.2
```

Note that this does not match `chap.12`. To match filenames `chap.10`, `chap.11`, and `chap.12`, use the notation

```
chap.1[012]
```

You can also place a hyphen (-) between two characters in brackets to denote a range. For example,

```
ls chap.[1-5]
```

is the equivalent of

```
chap.[12345]
```

A range of characters can also be indicated in brackets. The notation `[a-z]` matches any lowercase character, `[A-Z]` matches any uppercase character, and `[a-zA-Z]` matches any character, regardless of case.

To match anything *except* a certain character or range of characters, use the exclamation point inside the brackets. When the first character following the left bracket ([) is an exclamation character (!), any character *not* enclosed in the brackets is matched. For example,

```
[!b]
```

matches any filename composed of one letter, except a file named `b`.

None of these metacharacters will match the initial period at the beginning of special files such as `.profile`. These must be matched explicitly. Periods that do not begin a filename can be matched by metacharacters.

If you use these metacharacters and the shell fails to match an existing filename, it will pass the character on as an argument to the command. For example, if you have one file named `bb`, the command

```
echo ??  
prints  
bb
```

The command

```
echo ?  
prints  
?
```

5.2 Input and output redirection

An executing command may expect to accept input and create output, possibly including error output (error messages). In the A/UX system, there are default locations set for input and output:

- Standard input is taken from the terminal keyboard.
- Standard output is printed on the terminal screen.
- Standard error output is printed on the terminal screen.

These defaults can be changed using the following metacharacters (also called **redirection symbols**). The redirection metacharacters are a way of using file descriptors, described in detail in “Redirection With File Descriptors.”

- < A less-than sign followed by a filename “redirects standard input” (takes command input from a file or device other than the keyboard). For example,

```
mail ellen < note
```

uses a file named `note` instead of a message typed from the keyboard as the input to `mail`.

- > A greater-than sign followed by a filename “redirects standard output” (prints command output in a file or to a device other than the terminal screen). If a file by that name already exists, its previous contents are overwritten; otherwise a new file is created. For example,

```
sort file1 > file2
```

uses a file for the output of the `sort` command. When `sort` completes, `file2` contains the sorted contents of `file1`.

See “Input and Output” for information on redirecting standard error output using file descriptors.

- >> Two greater-than signs followed by a filename append the output of a command to a file. If no file by that name exists, one is created. For example,

```
who >> log
```

appends the output of the `who` command to the end of the existing file `log`.

5.3 Combining commands: pipelines

You can send the output of one command as input to another command by using the vertical bar (`|`). When two or more commands are joined by a vertical bar, the command line is called a **pipeline**.

For example, to see which files in a directory contain the sequence `old` in their names, you can use a pipeline as follows:

```
ls | grep old
```

The pipe character (`|`) tells the shell that output from the first command (the list of files produced by the `ls` command) should be used as input to the `grep` command. The output of the pipeline (filenames in the current directory containing the string `old`) prints on standard output (unless you redirect it to a file).

Pipelines may consist of more than two commands; for example,

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

prints the number of files in the current directory whose names contain the string `old`.

Pipelines may also be executed in the background. For example, to avoid the time-consuming process of waiting for a very large file to be sorted and printed, you could give the following pipeline:

```
sort mail.list | lp &
```

This pipeline would sort the contents of a file named `mail.list` and send the sorted information to the `lp` program to be placed on the printer queue. The shell would respond with the process ID of the last command in the pipeline.

The `tee` command is a “pipe fitting”; it can be put anywhere in a pipeline to copy the information passing through the pipeline to a file. See `tee(1)` in *A/UX Command Reference* for more information.

A **filter** is a program or a pipeline that transforms its input in some way, writing the result to the standard output. For example, the `grep` command finds those lines that contain some specified string and prints them as output.

```
grep 'correction' draft1
```

prints only the lines in `draft1` that contain the string `correction`.

Filters are often used in pipelines to transform the output of some other command. For example,

```
who | grep jon
```

prints

```
jon          ttyt8          Jul 21 12:25
```

if a user whose login name is `jon` is currently logged into the system on `tty01`.

5.4 Command grouping

You can use the following metacharacters to group commands together:

- `;` Group several commands on one command line by separating one command from another with a semicolon (`;`). The commands will be executed sequentially in the order in which they appear. For example, the command line

```
cd test; ls
```

changes to the `test` directory and then lists its contents.

- & Group background commands on a single line by separating them with ampersands (&) and then ending the line with another ampersand. The background commands will exit independently while the shell continues to accept new commands in the foreground.
- { } Use braces to group commands for functions and control-flow constructs (see “Defining Functions” and “Control-Flow Constructs”). You can also use braces to group the output from several sequential commands, which is then used as the input to a following command in a pipeline. Braces used in the latter way are recognized only when they are the first word of a command or are preceded by a semicolon or newline, and when the first brace is followed by a space. For example, to put the date and the list of users into one file (`log`), you could give the command

```
{ date; who; } > log
```

Note the space following the first brace and the semicolon following the last command in the braces; these are required. If you type a newline before closing with another brace, you will see the secondary prompt until you give the closing brace. Note that commands enclosed in braces are executed by the current shell (that is, a new instance of the shell is not invoked to execute them).

- () Enclose a group of commands in parentheses to execute them as a separate process in a subshell (a new instance of the shell). For example,

```
(cd test; rm junk)
```

first invokes a new instance of the shell. This shell changes the directory to `test` and then removes the file `junk`. After this, control is returned to the parent shell, where the current directory is not changed. Thus, when execution of the commands is over, you are still in your original directory.

The commands

```
cd test; rm junk
```

(without the parentheses) are executed in the current shell and have the same effect but leave you in the directory `test`.

5.5 Conditional execution

You can use the following symbols to indicate that your command should be executed only if some condition is met:

&& The command form

```
command1 && command2
```

means “If *command1* executes successfully (returns a zero exit status), then execute *command2*.”

| | The command form

```
command1 | | command2
```

does the reverse. This form means “If *command1* does not execute successfully (returns a nonzero exit status), then execute *command2*.”

For exit status, see “Exit Status: The Value of the Command.”

Conditional execution is also available in joining pipelines. For other ways of obtaining conditional execution, see “Control-Flow Constructs.”

5.6 Quoting

If you need to use the literal meaning of one of the shell metacharacters or control the type of substitution allowed in a command, use one of the following **quoting mechanisms**:

**** A backslash preceding a metacharacter prevents the shell from interpreting the metacharacter. For example, to use the A/UX `echo` command to display a question mark, you must precede the question mark with a single backslash (`\`). Thus,

```
echo \?
```

prints

```
?
```

Without the backslash, the `echo` command would generate a list

of all one-character filenames in the current directory. If there are none, the command returns

?

- ' ' Single quotes prevent the shell from interpreting any metacharacters in the enclosed string. The command

```
echo '*test'
```

prints

```
*test
```

while the command

```
echo *test
```

attempts to list all the files in your current directory ending with the characters `test`. If there are none, the command returns

```
*test
```

- " " Within double quotes, variable substitution and command substitution occur, but filename expansion and the interpretation of blanks do not. For example, if you have the variable `message1` with the value `"this is a test"`, the command

```
echo "$message1"
```

prints

```
this is a test
```

Double quotes can also be used to give a multiword argument to commands; for example,

```
echo "type a character"
```

For more information on variable substitution, see `"Positional Parameters and Shell Variables."` You can also suppress filename expansion universally by invoking the shell with the `-f` option; see `"Shell Execution Options."`

- ‘ ‘ A command name enclosed in back quotes is replaced by the output from that command. This is called **command substitution**. For example, if the current directory is `/usr/marylyn/bin`, the command

```
i=`pwd`
```

is equivalent to

```
i=/usr/marylyn/bin
```

If a back quote occurs within the command to be executed, you must escape it with a backslash (`\``); otherwise the usual quoting conventions apply within the command.

Command substitution takes place before the filenames are expanded. If the output of substituted command is likely to be more than one word, the command must be enclosed in double quotes as well as back quotes; for example,

```
a="`head -1`"
```

where the command `head -1` (read the first line of input) might yield more than one word.

6. Working with more than one shell

When you wish to use another A/UX shell, you can use one of the following commands:

`sh` This spawns another instance of the Bourne shell.

`ksh` This spawns an instance of the Korn shell.

`csh` This spawns an instance of the C shell.

You can type these at your shell prompt; for example,

```
csh
```

In this case, your new shell will run as a **subshell** or “child” of your current one. You can use the `exit` command or the `eof` sequence to return to your login shell whenever you wish. The **login shell** is the shell that is automatically invoked when you log in. (If you accidentally give the `exit` command or the `eof` sequence in your login shell, you will be logged out of the system altogether.)

6.1 Changing to a new shell

You can also obtain a new shell using the `exec` command; for example,

```
exec csh
```

If you use the `exec` command, the C shell program `csh` *replaces* your current shell. You cannot return to your original shell; it has disappeared.

Generating new instances of a shell affects the environment settings for each shell. See “The Environment and New Shell Instances” for more information.

6.2 Changing your default shell

To change your default shell from the Bourne shell to the Korn or C shell, use the `chsh` command. For example,

```
chsh login.name /bin/ksh
```

(where *login.name* is your login name on this system) changes your default login shell to the Korn shell. See `chsh(1)` in *A/UX Command Reference* for more information.

7. The environment

The **environment** is a list of variables and other data that is available to all programs (including subshells) invoked from the shell. A shell inherits the environment that was active when it started, and passes that environment (including any modifications) to all programs it invokes.

If you assign values to variables using the `set` command or the assignment operator (`=`) at the shell prompt (or within a shell script), these remain local to the shell in which you assigned them. If you use the `export` command (or set the `-a` shell option; see “Shell Execution Options”), these changes will be passed on to any subshells you invoke and to executing commands.

Note: Modifying the environment in a subshell (for example, in a shell script) never changes the parent shells or their environments. Because these changes are made to a *copy* of the parent shell’s environment, the parent shell’s environment is never affected by changes in a subshell, even if you use the

`export` command. When a subshell terminates, its environment no longer exists.

In general, the most essential variables are assigned default values during login or by the shell every time you invoke it. Convenient but inessential variables are simply left unassigned. Thus a default environment is created for you.

7.1 Listing existing values

The `env` command and the `printenv` command both list the values of all variables in the current environment.

The `export` command without an argument lists all explicitly exported variables in the environment. Variables with default values assigned by the shell, variables not exported in the current shell, and variables local to the current shell are not listed.

The `set` command without arguments lists the values of all variables in the current shell, including default values, values in the environment, local shell variables, and the text of all functions defined.

7.2 Assigning values to environment variables

Setting up your own customized environment is not necessary, but it can make your work easier and more efficient. To customize your working environment, you may change the default values assigned to some of the environment variables and add others that have not been included in the default environment.

Unless you have set the `-a` shell execution option (which tells the shell to export all variables automatically; see “Shell Execution Options”), the process of assigning a value to an environment variable requires two commands. The command syntax

```
name=value
```

sets a variable *name* to *value*. Note that there are no spaces around the equal sign; this is the required format. By convention, environment variables have uppercase characters in their names.

After you have assigned a value, the command syntax

```
export name
```

includes the variable *name* and the *value* you assigned to it in the environment for this shell. If you don't export the variable, the shell will not be able to pass it to your commands or programs.

Thus, the complete process of assigning a value to the USER variable would be

```
USER=daphne
export USER
```

7.3 Removing environment variables

The command

```
unset name
```

removes the specified variable. The PATH, PS1, PS2, MAILCHECK, and IFS variables cannot be removed.

7.4 Commonly used environment variables

The following variables are typically inserted into the environment. By convention, environment variable names are uppercase. Some of these variables are assigned default values at login or by the shell at invocation. All of them can be reset by the user.

HOME	This variable specifies your home directory. The login procedure sets the value of this variable to the pathname of your login directory.
CDPATH	The value of this variable should be a list of absolute pathnames of directories (separated by colons) that you use frequently. The shell uses this variable when you give an argument to the <code>cd</code> command that is not a relative or absolute pathname. This variable is usually set in the <code>.profile</code> file; otherwise its default value is the current directory.
EXINIT	This variable indicates various options for your editing environment when you are using the <code>ex</code> or <code>vi</code> text editing program (see "Using <code>ex</code> " and "Using <code>vi</code> " in <i>A/UX Text Editing Tools</i>).
PATH	The value of this variable should be a series of pathnames separated by colons (:). The shell uses the value of PATH executable programs whenever you give

a command. If the directory containing the command is not specified, the shell displays an error message. For example, if you enter the command `foo`, the shell prints

```
foo: not found
```

`PATH` is usually set in the `.profile` file. For efficiency, the list of directories in the `PATH` variable should be in order from the directories containing commands most often used to those least often used. The default value for `PATH` is the current directory, `/bin`, and `/usr/bin`.

- `MAIL` The shell uses this variable as the pathname of the file where your mail is delivered. This variable is typically set in the file `.profile` in the user's login directory.
- `MAILCHECK` This variable specifies how often (in seconds) the shell will check for the arrival of mail in the file specified in `MAIL`. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.
- `PS1` This variable specifies the primary prompt string (the prompt you see when the shell is waiting for you to give a command). The default setting is the dollar sign (`$`).
- `PS2` This variable specifies the secondary prompt string (the prompt you see when the shell is waiting for more information for a command you have already started). The default setting is the greater-than sign (`>`).
- `IFS` The shell uses this variable to interpret command strings. `IFS` stands for "Input Field Separator." The default values of this variable are space, tab, and newline, specifying the characters used to separate the parts of commands. You can reset this to include any data delimiters.
- `SHELL` This variable specifies your preferred login shell. It is set at login to the value found in the `/etc/passwd` file. The default shell is the Bourne shell. For instructions on how to change your login shell, see `chsh(1)` in *AUX Command Reference*.

TZ This variable indicates your time zone. It is set at login.

TERM This variable specifies the type of terminal you are using. The default value is `mac2`. You can find out what your current terminal type is with the command

```
echo $TERM
```

7.5 The environment and new shell instances

When you invoke a new instance of the shell (using the `sh` command for the Bourne shell), the values you have exported to the environment (using the `export` command) are *copied* to the environment of the new shell. If you have assigned values to variables without exporting them to the environment, these remain local to the parent shell. You may reset the value of any exported variable within the subshell. Because these changes are made to a *copy* of the parent shell's environment, the parent shell's environment is never affected by changes in a subshell, even if you use the `export` command. Note, however, that these changes will be passed on to new instances invoked from the subshell. When a subshell terminates, its environment no longer exists.

In the Bourne shell the `.profile` file is read only once, at login. Thus, if you have changed the value of an environment variable, the subshell will inherit the new value, not the value set routinely in `.profile`. You can force a new instance of the shell to read `.profile` by using the “dot” command (`.`); see “Executing Shell Scripts.”

In general, running one shell as the child of another (for example, running the C shell under the Bourne shell) will not cause any problems. The only exception may be if you have assigned values to environment variables that are significant to the other shell. See Chapters 3 and 4, “Korn Shell Reference” and “C Shell Reference.”

7.6 Special environments

Normally, the environment for a command is the complete environment of the shell where the command was given. You can change the environment used by a command in three ways:

- Augment the environment by inserting additional variables and new values into the environment. This is done by preceding the

command with one or more assignments to variables on the command line. For example,

```
a=b command
```

Note that because variable substitution occurs before the environment is changed, you cannot assign environment variables whose values are then immediately referenced on the command line. For example, the sequence of commands

```
x=5
x=3 echo $x
prints
5
not
3
```

because the value of `x` is inserted into the command line before the environment is changed.

- Set the `-k` shell option using the command

```
set -k
```

When `set`, this shell option inserts variables and values given on the command line into the environment for a particular command. For example, if the `-k` option is not set, the command

```
echo a=b c
prints
a=b c
```

After `-k` has been set, `a=b` is interpreted as a variable assignment instead of an argument, and the same command prints

```
c
```

Note that because values are substituted for variables before the environment is changed, this is subject to the same limitation documented above.

- Use the A/UX command

```
env [-][name=value ...][command][args]
```

to set the environment for the command. With this command, you can not only add things to the environment inherited by a command, but also exclude the current environment. To add variables and their values to the current environment, give the variables and values before the command name. For example, to run a subshell with a changed `PATH` environment variable, you could give the command

```
env PATH=directory-list sh
```

For the duration of the new shell (and its subshells), the `PATH` variable would be set to the directories in the list.

To set up a completely new environment, first give the option `-`, which excludes the current environment, and then assign the variables and values you want. These (and only these) will be available in the environment for the new command.

7.7 The default environment on your system

Whenever you log in, the following procedures occur:

- The `login` program sets the default `PATH` and sets values for the variables `HOME`, `LOGNAME`, and `SHELL` from the information in the system file `/etc/passwd`.
- The login shell then checks the file `/etc/profile` to find out the default environment to set up for all users. This file may contain settings for `PATH`, `TZ`, and `TERM`.
- The login shell assigns default values to `PS1` (the primary prompt), `PS2` (the secondary prompt), `MAILCHECK`, and `IFS` (Input Field Separator).

When you invoke new instances of the shell (for example, using the `sh` command), the new shell checks the environment for any new values of these variables you may have placed there. If it doesn't find any values in the environment, it assigns the default values.

Then the new shell reads your `.profile` file. If you have assigned new values there, it uses your values instead of the defaults.

8. The .profile file

The `.profile` file is simply a text file. It contains a series of commands typed exactly as you would type them at the shell prompt. Every time you log in, the shell looks in your home directory for a file named `.profile` and executes all the commands found there before issuing the shell prompt and taking commands. If no `.profile` file exists, your environment will simply be the default environment created by the shell at login.

8.1 A sample .profile file

The following is a sample `.profile` file:

```
PATH=/bin:/usr/bin:/users/elaine/bin:$HOME
export PATH
CDPATH=/users/group.project/elaine/revisions
export CDPATH
MAILCHECK=0
export MAILCHECK
EXINIT='set wm=10'
export EXINIT
date
ls
```

The variables and commands in this file are discussed in the sections that follow.

8.1.1 Locating commands

The `PATH` environment variable lists the directories (separated by colons) where the shell will look for the executable files that are A/UX (or user-defined) commands. Each time you give a command, the shell searches the directories listed in the order specified. Most A/UX commands are located in the `/bin`, `/usr/bin`, or `/usr/ucb` directory. When you assign a value to `PATH`, be sure to include these directories.

If the shell cannot find the file in one of the directories specified, the command cannot be executed and you will see the message

```
command-name: not found
```

The directories listed in the `PATH` variable are specified by their absolute pathnames, separated by colons. If the list of directories

begins with a colon, the path search begins in the current directory. At login, the `PATH` variable is set as follows:

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

This assignment sets the `PATH` variable to the current directory and the system directories `/bin` and `/usr/bin`.

To reset the `PATH` variable in `.profile`, insert the lines

```
PATH=:/bin:/usr/bin:/usr/ucb:/users/name/bin:$HOME
export PATH
```

The `export` command is discussed under “Customizing Your Environment.”

If you include the pathnames of personal directories that contain shell programs you have written, these will be accessible to the shell no matter what your current directory is. If you wish to execute a command or shell program that is not in one of the directories in your `PATH` variable, simply give the absolute pathname of the directory where the command or shell program is to be found.

For information on referencing variables using the `$` syntax (as in `$HOME` above), see “Parameter and Variable Substitution.” For more information about pathnames, see the glossary in *A/UX System Overview*.

8.1.2 Shortcuts in changing directories

If `CDPATH` is set, you can use the `cd` command with a simple directory name that is neither an absolute nor a relative pathname. The shell then searches for that directory in all the directories listed in `CDPATH`. The directories are searched in the order specified. If `CDPATH` is not set, only the current directory is searched.

If the directory you specify, for example `tmp`, is not found in any of the directories given in `CDPATH`, you will see the message

```
tmp: bad directory
```

After `CDPATH` is set, you can still, of course, give the relative or absolute pathname of any directory you wish. When you give an absolute or relative pathname in the `cd` command, `CDPATH` is not used.

8.1.3 Receiving mail

The `MAILCHECK` environment variable specifies how often (in seconds) the shell should check for new mail. When you log in, the shell sets `MAILCHECK` to 600 seconds (10 minutes). You can change this to whatever you wish using the commands

```
MAILCHECK=0
export MAILCHECK
```

These commands assign and export the value of the `MAILCHECK` as 0. When `MAILCHECK` is 0, the shell checks for new mail before each prompt.

8.1.4 Your editing environment

The `EXINIT` environment variable tells the shell how to initialize the `vi` or `ex` editing program. This variable is set to a series of editor commands that should be run every time the editor is called before any commands are read from the terminal. In the sample `.profile` above, for example, the commands

```
EXINIT='set wm=10'
export EXINIT
```

assign and export the value of `EXINIT` as the command

```
set wm=10
```

which sets the word-wrap margin so that the editor will automatically break lines ten spaces before the right margin. The command is enclosed in double quotes because the entire string must be treated as one “word” and not divided up.

For details on `EXINIT`, see *AUX Text Editing Tools*. For the use of double quotes, see “Quoting.”

8.2 Customizing your login procedure

You can also use your `.profile` file to customize your login procedure. In the sample `.profile` above, the commands

```
date
ls
```

direct the shell to display the date and time and then list all the files in the current directory before displaying the shell prompt. These will be

executed at login.

You can include any commands you wish in `.profile`, including your own functions and shell scripts. One commonly included command is the `sh` command. If you include the `sh` command, the shell will invoke the shell-layering process before you give any commands from the terminal. This means that instead of the normal shell prompt, your first prompt will be the `sh` prompt:

```
>>>
```

For information on shell layering, see Chapter 5, “Shell Layering.”

9. Shell execution options

The shell is a program like other A/UX commands, and it too has a variety of options used to control how it executes. All shell execution options can be set using the `set` command as follows:

```
set -opt[opt...]
```

Or they can be specified on the command line when you invoke a new shell or run a shell script with the `sh` command:

```
sh -opt[opt...] name
```

Use the `set` command to set new options in your current shell. Use the `sh` command to invoke a subshell with the options specified or to run a script with options.

To turn options off, precede the option with a plus (+) instead of a minus (-).

The variable `$-` contains a list of all the options set. For example, if you have the `a` and `x` shell execution options set, the command

```
echo $-
```

returns

```
ax
```

9.1 Options that affect the environment

- a When the `-a` shell option is set, all variable assignments result in that variable and its value being inserted into the environment.

You do not need to use the `export` command to insert new values.

- k The shell execution option `-k` can be used to insert variables and values into the environment for a particular command; see “Special Environments.”

9.2 Options for invoking new shells

In addition to the options available with the `set` command, there are four options that can be used only when a new shell is invoked with the `sh` command. These are

-c *string*

If the `-c` flag is present, *string* is executed. After execution, control is returned to the parent shell. This command is often used to execute shell scripts; see “Executing Shell Scripts.”

-s If the `-s` flag is present or if no arguments remain, commands are read from the standard input.

-i If the `-i` flag is present, the shell is interactive. The terminate signal is ignored (so that `kill 0` does not kill an interactive shell), and the *interrupt* signal is caught and ignored (so that `wait` is interruptible). In all cases, the *quit* signal is ignored by the shell.

-r If the `-r` flag is present, the shell invoked is a restricted shell. Restricted shells cannot change directories, alter the value of the `PATH` environment variable, redirect output, or specify path or command names containing the symbol `/`. See “Restricted Shell.”

During shell invocation, if the first character of argument 0 is a `-`, commands are read from the `.profile` file.

10. Restricted shell

The Bourne shell supports a limited version called the **restricted shell**, or `rsh` (note that in A/UX, the 4.2 BSD `rsh` remote shell network program has been renamed `remsh` to prevent conflict with this program).

This version of the shell is used to set up login names that have restricted access to the file system (they cannot execute the `cd` command or redirect output) and a limited menu of commands (they cannot specify absolute pathnames or change the value of their `PATH` variable).

This is useful if you want to allow several users to log into your machine but want to restrict them to a single directory or to a limited subset of commands. In that case, you may want to set up a special directory of commands (for example, `/usr/rbin`) that can be safely invoked by all users, and include only that directory in the value of the `PATH` variable. Because `rsh` is invoked after `.profile` is read, you can set up such an environment by writing special `.profile` files for such users. See `sh(1)` in *A/UX Command Reference* for more information.

11. Using shell layering

The `shl` program allows you to create up to seven labeled subshells called **shell layers** within your login shell. These layers can then be referred to by name (or number), suspended and resumed, deleted, and so on. Each of these layers appears like your login shell, but can be used to run a process while you switch to another layer. This provides a management scheme for multiple concurrent processes. See Chapter 5, “Shell Layering.”

12. Overview of shell programming

A shell program is simply a list of commands. These commands can be entered at the prompt or inserted in a file. They may contain

- variables and assignments
- control-flow statements (for example, `if`, `for`, `case`, or `while`)
- built-in shell commands
- any A/UX command

Input for the shell program can be read from the keyboard (this is the default standard input), taken from files, or embedded in the program itself (using here documents).

Shell programs can write output to the terminal screen (the default standard output), to files, or to other processes (via pipes).

When the shell program executes, each command is executed until the shell encounters either an *eof* character or a command delimiter that directs it to stop. During execution, you can trap errors and take appropriate action.

Shell program variables are strings. Arithmetic is not provided, but is available indirectly using the `expr` command.

12.1 Writing shell programs

You can enter a shell program at the prompt. When you use a built-in shell command that expects a delimiter (such as `done`) or a certain type of input, the secondary shell prompt appears after you press RETURN. This prompt (`>` by default) appears at each line until you give the expected delimiter; for example,

```
$ for i in *
> do
> cat $i
> done
$
```

Note that you can send an *interrupt* to cancel the script and return to the primary prompt.

You can also write a shell program in a text file (using a text editor) and then execute it (see “Executing Shell Scripts”). These program files are often called **shell scripts**. Note that all shell programs may be entered at the shell prompt or inserted in a file. This does not affect their actions. Hereafter “shell scripts” will be used to refer to shell programs that reside in a file.

12.2 Executing shell scripts

There are several ways to execute a shell script; these differ mostly in terms of which instance of the shell is used for the execution.

- You can use the `sh` command to read and execute commands contained in a file. The script will be run in a subshell, which means that it will have access only to the values set in the environment and will be unable to alter the parent shell. The command

```
sh filename args ...
```

causes the shell to run the script contained in *filename*, taking the *args* given as positional parameters. Shell scripts run with the `sh` command can be invoked with all the options possible for the `set` command.

- You can change the mode of the shell script file to make it executable. For example,

```
chmod +x filename
```

makes *filename* executable. Note that you may want to modify your `PATH` variable to include a personal directory containing your shell scripts. When you have done this, you can use your script names as ordinary commands, regardless of your current location in the file system. Then the command

```
filename args ...
```

has the same effect as using the `sh` command. The arguments become the positional parameters (see “Positional Parameters”); the script is run in a subshell, which means that it will have access only to the values set in the environment and will be unable to alter the parent shell.

- You can run a shell script inside the current shell by using the “dot” command (`.`). The dot command tells the current shell to run the script; no subshell is invoked. This should be used if you wish to use local shell variables or functions, or modify the current shell:

```
. filename args ...
```

Note that there must be a space between the dot and the filename. Because the commands are executed in the current shell, this is the way to run a script that is to change values in the shell. The arguments become positional parameters. Otherwise the positional parameters are unchanged.

- You can run an executable shell script with the `exec` command (the file containing the shell script must have execute permission). This should be used when the shell script program is an application designed to execute in place of the shell and

replace interaction with it:

```
exec filename args ...
```

In this case, the shell script *replaces the current shell*. This means that when the script is over, control will not return to the shell. If you were in a login shell, you will be logged out.

12.3 Comments

A word beginning with a number sign (#) causes that word and all the following characters up to a newline to be ignored.

12.4 Writing interactive shell scripts

A shell script can invoke an interactive program such as the `vi` editor. If standard input is attached to the terminal, `vi` will read commands from the terminal and execute them just as if invoked from an interactive shell. After the session with `vi` is finished, control will pass to the next line in the script. In a similar manner, a script can invoke another copy of a shell (using `sh`, `csh`, or `ksh`), which will interpret commands from the terminal until you send an *eof*. Control will be returned to the script. You can use this to create a special environment for certain tasks by setting environment variables in a shell script and then invoking a new subshell.

You can also write interactive shell scripts by using the `read` and `eval` commands and prompting users to enter commands:

```
read command
eval $command
```

The first line will read the user's command line into the variable `command`. The `eval` command will then cause the command to execute.

12.5 Canceling a shell script

You can cancel a shell script just like an ordinary A/UX command. If the script is running in the background, use the A/UX `kill` command. See "Canceling Commands" for details on `kill` and various types of interrupts that can stop a command.

Note: Interrupts can be trapped and handled within the script using the `trap` command. See "Summary of Bourne Shell Commands."

12.6 Writing efficient shell scripts

In general, built-in commands execute more efficiently than A/UX commands. See “Summary of Bourne Shell Commands” at the end of this chapter for a complete list of these commands. The following built-in commands are useful in constructing efficient shell scripts:

- `hash` This causes the shell to remember the search path of the command named.
- `ulimit` This can be used to set a limit on files written by processes.
- `times` This prints the accumulated user and system times used by the current shell.

You can also set the `-h` shell execution option using

```
set -h
```

This will locate and remember functions as they are defined, instead of when they are invoked, as normal.

Careful setting (or resetting inside a shell script) of the `PATH` and `CDPATH` environment variables make sure that the most frequently used directories are listed first. This also improves efficiency.

13. Command evaluation

When you give a command, the shell evaluates the command in one pass and then executes it. To force more than one pass of evaluation, use the `eval` command (see “Forcing More Than One Pass of Evaluation”).

While evaluating the command, the shell performs the following substitutions on variables:

- **Variable substitution.** This replaces variables preceded with `$` (for example, `$user`) with their values. Only one pass of evaluation is made. For example, if the value of the variable `user` is `daphne`, then the command

```
echo $user  
prints  
daphne
```

However, if the value of the variable `user` is `$name`, then the command

```
echo $user
```

prints

```
$name
```

The second variable is never evaluated, and the value is not substituted. See “Parameter and Variable Substitution” for more information.

- **Command substitution.** The shell replaces a command enclosed in back quotes with the command’s output. For example, if the current directory is `/users/doc/virginia`, then the command

```
echo `pwd`
```

prints

```
/users/doc/virginia
```

- **Blank interpretation.** The shell breaks the characters of the command line into words separated by delimiters (called “blanks”). The delimiters that are interpreted as blanks are set by the shell variable `IFS`; by default, they are spaces, tabs, and newlines. The null string is not regarded as a word unless it is quoted; for example,

```
echo ''
```

passes the null string as the first argument to `echo`, whereas the commands

```
echo
```

and

```
echo $null
```

(where the variable `null` is not set or set to the null string) pass no arguments to the `echo` command.

- **Filename expansion.** The shell scans each word for filename expansion metacharacters (see “Using Shell Metacharacters”)

and creates an alphabetical list of filenames that are matched by the pattern(s). Each filename in the list is a separate argument. Patterns that match no files are left unchanged.

These evaluations also occur in the list of words associated with a `for` loop.

13.1 Forcing more than one pass of evaluation

Sometimes more than one pass of evaluation is necessary for a command to be interpreted correctly. For example, suppose that the following two lines occur near the beginning of a shell script:

```
name=elaine
err_33='echo $name: user not found'
```

If you give the command

```
$err_33
```

you get

```
$name: user not found
```

(which is not quite what you want). In cases like this, you can use the built-in command `eval`. So, the command

```
eval $err_33
```

forces two evaluations of the value of the variable `err_33`. Thus it prints

```
elaine: user not found
```

In general, the `eval` command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed.

13.2 Command execution

After all substitution has been carried out, commands are executed as follows:

- Built-in commands, functions, and shell scripts run with the dot command (`.`) are executed in the current shell. The command has available all current shell execution options, the values of variables and environment variables, and functions defined in the current shell.

- A/UX commands, programs, executable shell scripts, shell scripts run with the `sh` command, and series of commands enclosed in parentheses are executed in a subshell. The current shell invokes a child shell that executes the commands and then returns control to the parent shell. Only the values in your environment are available to these processes.
- Commands and executable scripts run with the `exec` command execute in place of the current shell.

If the A/UX command or program name does not specify a pathname, the environment variable `PATH` is used to determine what directories should be searched for the command. The only exception to this is built-in commands.

For more information about the execution of shell scripts, see “Executing Shell Scripts.”

13.3 Exit status: the value of the command

If a command executes successfully, its exit value is usually zero (0). If it terminates abnormally, its exit value is often nonzero. The shell saves the exit value of a command. These are used primarily in shell scripts.

To check the exit status of a command, use the command

```
exit $?
```

See “Parameters and Variables Set by the Shell” for more information. See the manual entry for the command in question in *A/UX Command Reference* or *A/UX System Administrator’s Reference* for exit status values.

14. Defining functions

You can use a **function definition** to assign a name to a command or list of commands. After you have defined a function, typing the function *name* (and any required arguments) causes the commands in *command-list* to be executed by the current shell.

The form of a function definition can be

```
name () { command-list; }
```

or

```
name () {  
  command-list  
}
```

The first brace ({) must be followed by a space or newline; the second must be preceded by a semicolon or newline. There cannot be a semicolon between the parentheses and the first brace.

For example, a function maintaining a daily log of users could be written as follows:

```
users () { date>>log; who>>log; }
```

or

```
users () {  
  date>>log; who>>log  
}
```

The function would first append the date and then the listing provided by the `who` command to the file named `log`.

Functions are commonly defined in the `.profile` file, although they can also be defined at the terminal or in a shell script.

Functions execute in the *current shell*, not in a subshell. During execution, any arguments become the positional parameters. After execution, they are reset to their former values, if any. This means that if a function is defined and used inside a shell script, the parameters of functions will not conflict with the parameters of the script.

Because they are executed in the current shell, functions share their variables with this shell and can create, alter, and assign shell and environment variables, as well as create new environment variables via `export`. Functions themselves, however, cannot be exported. This means that they are available only in the shell where they were defined (for example, the login shell if they are defined in the `.profile` file) and that a function defined in a particular shell will be available only to shell scripts run with the dot command (`.`) in that shell.

The `return` command in a function definition causes a function to terminate with the exit status specified by *n*. For example,

```
users () {
date>>log;who>>log
return 1
}
```

causes the function to terminate normally with a return value of 1. If the *n* is omitted from the `return` command, the exit status is that of the last command executed.

To speed up execution of functions, you can set the `-h` shell option:

```
set -h
```

This option causes the shell to remember the location of the function when it is defined rather than locating it every time it is executed.

To list the text of the defined functions, use the `set` command without arguments. (This will list the values of all variables currently set in the shell, including functions.) To remove a function, use the `unset` command followed by the name of the function.

15. Positional parameters and shell variables

A shell script may use two types of variables:

Positional parameters: These are string variables referred to by the numbers [0–9]. These numbers refer to the position of the parameter on the command line. Positional parameters are set on the command line and contain the arguments to the script. If more than ten positional parameters are required, the `shift` command can be used to discard old values.

Shell variables: These are string variables referred to by name. They may be assigned on the command line or inside the script itself.

The relationship between variables inside a shell script and existing shell variables depends on how the script is run. See “Executing Shell Scripts.” In all cases, shell scripts have access to the variables and values in the environment.

15.1 Positional parameters

Positional parameters may be referred to by the numbers [0–9] and set as arguments on a command line. When you enter a command at the prompt, the shell stores the elements of the command line in

parameters: the command name is stored in parameter 0, the first argument is stored in parameter 1, the second argument in parameter 2, and so forth. Thus, for the command

```
diff letter1 letter2
```

parameter 0 is `diff`, parameter 1 is `letter1`, and parameter 2 is `letter2`. For the command

```
echo "not a directory"
```

parameter 0 is `echo` and parameter 1 is `"not a directory"`.

A shell script may refer to parameters by number; for example,

```
echo $1  
echo $2
```

These will be substituted by the arguments given in that position on the command line; for example, for the command

```
myscript arg1 arg2
```

parameter 0 is `myscript`, parameter 1 is `arg1`, and parameter 2 is `arg2`. This prints

```
arg1  
arg2
```

15.1.1 Setting values in a script

The `set` command creates a new sequence of positional parameters and assigns them values. After execution, all the old parameters are lost. For example, the command

```
set *
```

creates a sequence of positional parameters set to the names of the files in the current directory (parameter 1 is the first filename, parameter 2 is the next filename, and so on). A subsequent command,

```
set hi there
```

creates new positional parameters, discarding the old values. This time there will be only two values set; the other positional parameters will have no values. A subsequent command,

```
echo $2 $1
```

displays

```
there hi
```

The command

```
echo $3
```

would have no effect, because there is no longer a parameter 3.

To set a positional parameter to a string of words separated by blanks, the entire string must be enclosed in double quotes. For example,

```
set "this is one positional parameter"
```

sets this entire string to the first positional parameter. Without the quotes, the phrase would be set, one word at a time, to the first five positional parameters.

Because the `set` command creates a new series of parameters, it is impossible to set only one parameter in a series. If only one parameter is set, it will be the first, and the remaining parameters will be lost.

The `set` command can also be used within a script to create positional parameters if none are given on the command line. Such parameters can then be used as a one-dimensional array.

After the `set` command is used to reset positional parameters, the internal shell variable `#`, which contains the number of positional parameters, is reset to reflect the new number of parameters. For details on the internal shell variables, see ‘Parameters and Variables Set by the Shell.’

15.1.2 Changing parameter positions

The `shift` command shifts positional parameters one or more positions to the left, discarding the value in the first position(s). The syntax is

```
shift [n]
```

If *n* is omitted, it defaults to 1. If *n* is specified, the shift takes place at the position *n*+1. For example,

```
shift 6
```

moves parameter 7 into position 1, parameter 8 into position 2, and so on, discarding the values that were stored in positions 1 through 6.

This can be useful, for example, when working through a list of files. After each file is processed, a shift can be performed, letting the next filename become parameter 1.

15.1.3 Number of parameters

The current number of positional parameters is stored in the system-maintained variable `#`. See “Parameter and Variable Substitution” and “Parameters and Variables Set by the Shell.”

15.2 Shell variables

Shell variables are named string variables. These variables can be assigned values anywhere in the script or on the command line. Variable names begin with a letter and consist of letters, digits, and underscores. Environment variables, which we have already encountered, are simply special kinds of shell variables (namely, shell variables that are available to all subshells).

15.2.1 Assigning values

Shell variables are assigned values with the syntax

```
name=value [name=value] ...
```

Note that there cannot be spaces surrounding the equal sign; this is required.

All values are stored as strings. Pattern-matching is performed. To set a variable to a string of words separated by blanks, the entire string must be quoted; for example,

```
longvar="this is a long variable"
```

After the variable assignments

```
user="fred stone" box='???' acct=18999
```

the following values are assigned:

```
user   = fred stone
box    = ???
acct   = 18999
```

Because the Bourne shell supports only string variables, all of these values (including 18999) will be strings of characters. Note that the question mark metacharacters must be quoted with single quotes to prevent pattern matching.

A variable may be set to the null string with the syntax

```
variable=
```

Shell variables may be set at the shell prompt to provide abbreviations for frequently used strings; for example,

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

moves *file* from the current directory to the directory */usr/fred/bin*.

An argument to a shell program of the form *name=value*, which precedes the command name, causes *value* to be assigned to *name* before execution begins. The value of *name* in the invoking shell is not affected. For example,

```
user=fred command
```

will execute *command* with *user* set to *fred*.

After variable assignments, any additional arguments are assigned to the positional parameters.

The *-k* flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. See “Special Environments.”

15.2.2 Removing shell variables

You can remove shell variables by using the *unset* command followed by the name of the variable:

```
unset name
```

The variable and its value will be removed.

15.3 Setting constants

Names whose values are intended to remain constant may be declared read-only. The form of this command is

```
readonly name ...
```

Subsequent attempts to assign values to read-only variables are illegal.

15.4 Parameter and variable substitution

Positional parameters and shell variables are referenced and their values are substituted when the identifier (the variable name or positional parameter number) is preceded by a dollar sign (\$):

```
$identifier
```

For example,

```
$j1 $1 $8 $version
```

For variables, *identifier* can be any valid name; for positional parameters, *identifier* must be a digit between 0 and 9 inclusive. Additional positional parameters must be moved into this range with the `shift` command described above, referenced with the `$*` notation described below, or accessed through the `for` construct.

Another notation for substitution uses braces to enclose *identifier*:

```
echo ${identifier}
```

This is equivalent to *\$identifier*. Braces are generally used when you may want to append a letter or digit to *identifier*. For example,

```
tmp=/tmp/ps  
ps a >${tmp}a
```

substitutes the value of the variable `tmp` and directs the output of `ps` to the file `/tmp/psa`, whereas

```
ps a >$tmpa
```

causes the value of the variable `tmpa` to be substituted.

A special shell parameter, `*`, can be used to substitute for all positional parameters (except 0, which is reserved for the name of the file being executed). The notation `@` is the same as `*` except when it is quoted. Thus,


```
echo "$*"
```

prints all values of all the positional parameters, and

```
echo "$@"
```

passes the positional parameters, unevaluated, to `echo` and is equivalent to

```
echo "$1" "$2" ...
```

15.5 Testing assignment and setting defaults

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

```
echo $d
```

or

```
echo ${d}
```

prints a blank line.

The following structures allow you to test whether variables or parameters are set and not null, and provide default values or messages. In these structures, *string* is evaluated only if it is to be substituted (command substitution, another variable, and so forth). If the colon is omitted, the shell checks only that the variable has been set; no action is taken if the variable or parameter is currently null.

```
${identifier:-string}
```

If the parameter or variable whose name is represented by *identifier* is set and is non-null, substitute its value; otherwise substitute *string*. The value of the variable or parameter is *not* changed. For example, if the variable `test` is null or unset, then

```
${test:-unset}
```

returns the string `unset`; otherwise the value of `test` is returned.

```
${identifier:+string}
```

If *identifier* is set and is non-null, substitute *string*; otherwise substitute nothing. The value of the variable or parameter is not changed. For example, if the variable `test` was null or unset, then

```
$(test:+unset)
```

returns nothing.

```
$(variable:=string)
```

If *variable* is not set or is null, set it to *string*; then substitute the new value. Positional parameters may not be assigned in this way. For example,

```
$(HOME:=/user/doc)
```

tests the environmental variable `HOME` to see if it had a non-null value. If it did not, it would be assigned the value `/user/doc` and this value would be substituted. Otherwise the original value of `HOME` would be returned.

```
$(identifier: ?string)
```

If *identifier* is set and is non-null, substitute its value; otherwise print *string* and exit from the shell. If *string* is omitted, the message

```
filename:identifier:parameter null or not set
```

prints. For example, a shell script named `distribute` that requires the parameter `directory` to be set might start as follows:

```
echo ${directory:? "distribution directory not set"}
```

If `directory` was not set, the script would immediately exit with the message

```
distribute:directory:distribution directory not set
```

15.6 Parameters and variables set by the shell

Except for the exclamation mark (`!`), the following parameters are initially defined by the shell; the `!` is defined only after a background task is executed. These parameters can be referenced anywhere *identifier* or *variable* appears in the standard forms described in the previous section; for example `echo $?`.

- ? The exit status of the last command as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise a nonzero exit status is returned. This is used in the `if` and `while` constructs for control of execution.

- # The number of positional parameters in decimal. For example, this notation is used in a script to refer to the number of arguments. An example of this use appears in the `case` section.
- * All the positional parameters (arguments) of a shell script. For example,


```
for i in $*
do
    echo $i
done
```

The above shell subroutine prints all the positional parameters.
- \$ The process ID of this shell in decimal. Because process IDs are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example,


```
ps a > /tmp/ps$$
command-list
rm /tmp/ps$$
```
- ! The process ID of the last process run in the background.
- The current shell flags, such as `-x` and `-v`.

16. Control-flow constructs

The shell has a variety of ways of controlling the flow of execution. The actions of the `for` loop and the `case` branch are determined by data available to the shell. The actions of the `while` or `until` loop and “if then else” branch are determined by the exit status returned by commands or tests. Control-flow constructs can be used together and loops can be nested.

In the following constructs, reserved words like `do` and `done` are only recognized following a newline or semicolon. *command-list* is a sequence of one or more simple commands separated or terminated by a newline or a semicolon.

16.1 `for` loops

To repeat the same set of commands for several files or arguments, use the `for` loop:

```
for name in word1 word2
do
    command-list
done
```

Note: The words `for`, `do`, and `done` must follow a newline or semicolon.

An example of such a procedure is `tel`, which searches a file of telephone numbers, `/usr/lib/telnet`, for the various names given as arguments to the command and passed as positional parameters. The text of `tel` is

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

Note that the “`for i`” notation is shorthand for “`for i in $*`”.

The command

```
tel fred
```

sets `i` to the name `fred` and prints those lines in the file `/usr/lib/telnet` that contain the string `fred`. It is equivalent to

```
for i in fred
do
    grep $i /usr/lib/telnet
done
```

The command

```
tel fred bert
```

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

To terminate a loop before the condition fails (or is met), or to continue a loop and cause it to reiterate before the end of *command-list* is reached, use the loop-control commands:

```
break [n]
continue [n]
```

These commands can appear only between the loop delimiters `do` and `done`. The `break` command terminates execution of the current loop; execution resumes after the nearest `done`. The `continue` command causes execution to resume at the beginning of the current loop.

For both `break` and `continue`, the optional *n* indicates the number of levels of enclosing loops at which execution should resume or continue. For example, the `break 2` in

```
for i in 0 1
do
  for j in 0 1
  do
    for k in 0 1 2 3
    do
      echo $i$j$k
      break 2
    done
  done
done
```

causes execution to resume two levels above the current loop.

16.2 case statements

The form of the `case` statement is

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

Each *command-list* except the last must end with “`;`”. (The semicolons after the last *command-list* are optional.) This breaks out of the `case` statement after execution. After execution of *command-list*, the `case` is complete and control passes to the command following `esac`.

Patterns may include filename expansion metacharacters. However, unlike filenames, the initial dot, slashes, and a dot following a slash do

not have to be matched explicitly. Different patterns to be associated with the same *command-list* are separated by the OR operator, the vertical bar (|). To be used literally, pattern-matching metacharacters must be quoted. Because an asterisk (*) matches any sequence of characters, it can be used to set up a default case. However, be careful in setting up the default; there is no check to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the next example, the commands following the first (*) will never be executed because the first (*) matches everything it receives.

```
case $# in
  *) exit ;;
  0) echo "no arguments given"
     exit ;;
esac
```

The following is an example of a case statement within a script named `append` that appends files:

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

When called with one argument, as in

```
append file
```

the system-set variable `#` is assigned the value 1 (the number of parameters in the call); and the standard input is appended (copied) onto the end of `file` using the `cat` command:

```
append file1 file2
```

Then the value of `#` is 2 and the command appends the contents of `file1` onto `file2`. If the number of arguments supplied to `append` (that is, the value of `#`) is greater than 2 or is 0, then the shell prints an error message indicating proper usage.

The following example illustrates the use of alternative patterns separated by a vertical bar (|):

```
case $i in
  -x|-y) command;;
esac
```

The same effect could be had by using the bracket metacharacters ([and]), as in

```
case $i in
  -[xy]) command;;
esac
```

When using metacharacters, the usual quoting conventions apply so that

```
case $i in
  \?) echo "input is ?" ;;
...
esac
```

matches the character ? for the first pattern.

A common use of the `case` construct is to distinguish between different forms of an argument. The following example is a fragment of a script that uses a `case` statement inside a `for` loop:

```
for i
do
  case $i in
    -[ocs]) ... ;;
    -*) echo "unknown flag $i" ;;
    *.c) /lib/c0 $i ... ;;
    *) echo "unexpected argument $i" ;;
  esac
done
```

16.3 while loops

The `while` and `until` commands cause the program to loop depending on whether or not a certain condition is met.

A `while` loop has the form

```
while command-list1
do
    command-list2
done
```

Note: The words `while`, `do`, and `done` must follow a newline or semicolon.

The `while` command tests the exit status of the last simple command in *command-list1*. Each time round the loop, *command-list1* is executed. If the last command executes successfully (a zero [true] exit status is returned), then *command-list2* is executed; otherwise the loop terminates. If the last command executes successfully but returns a nonzero exit status, the `while` loop will think it is false and terminate. For example, the script

```
while test $1
do
    command-list
    shift
done
```

loops through all the positional parameters. For each iteration of the loop, the `test` command is used to determine if the parameter exists. If it does, then `test` returns a zero (true) exit status and the following commands execute.

The `shift` command is used to rename the positional parameters `$2`, `$3`, ... as `$1`, `$2`, ..., and remove the first one, `$1`. This entire loop is equivalent to

```
for i
do
    command-list
done
```

To create an endless unconditional `while` loop, use the A/UX `true` command, which always returns a zero exit status.

16.4 `until` loops

The `until` loop has the form

```
until command-list1
do
  command-list2
done
```

Note: The words `until`, `do`, and `done` must follow a newline or semicolon.

It works the same way as a `while` loop, except that the termination condition is reversed. Each time round the loop, *command-list1* executes; if the last command does *not* execute successfully (returns a nonzero [false] exit status), then *command-list2* is executed.

A common use for an `until` loop is to wait until some external event occurs and then run some commands. For example,

```
until test -f file
do
  sleep 300
done
commands
```

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

To terminate a loop before the condition fails (or is met), or to continue a loop and cause it to reiterate before the end of the command list is reached, use the loop-control commands:

```
break [n]
continue [n]
```

These commands can appear only between the loop delimiters `do` and `done`, as in the `for` loop. See “`for` Loops” for more information on using the `break` and `continue` commands.

For both `while` and `until` loops, the exit status of the loop is that of the last command executed in *command-list2*. If no commands in

command-list2 are executed, then a zero exit status is returned.

To create an endless nonconditional `until` loop, use the `A/UX false` command. See `true(1)` in *A/UX Command Reference* for details.

16.5 **if then else**

The form of the “if then else” conditional branch is

```
if command-list1
then
    command-list2
[else
    command-list3]
fi
```

In this structure, `else` and *command-list3* are optional. The `if` command tests the exit status of the last simple command in *command-list1*. If the last command executes successfully (a zero [true] exit status is returned), then *command-list2* is executed; otherwise *command-list3*, if present, is executed. For example, the `if` command can be used with the `test` command to test for the existence of a file, as below:

```
if test -f file
then
    command-list1
else
    command-list2
fi
```

People find it natural to name test files `test`, which makes it awkward (and dangerous) to use the `test` command as well. A harmless alternative is the `[]` construct:

```
if [-f file]
then
    command-list1
else
    command-list2
fi
```

Multiple conditions can be tested with a nested `if` command:

```
if condition1
then
    command-list1
else
    if condition2
    then
        command-list2
    else
        if condition3
        command-list3
        fi
    fi
fi
```

Note that each of the nested `if` commands requires its own `fi`. You can also use a single `if` construct to achieve this effect:

```
if condition1
then
    command-list1
elif condition2
then
    command-list2
elif condition3
    command-list3
fi
```

Note that this is a single `if` construct with only one terminating `fi`.

An example of the `if` statement can be found in the following script, which updates the last modified time for a list of files.

```

flag=
for i
do
    case $i in
        -c) flag=N ;;
        *) if test -f $i
            then
                touch $i
            elif test $flag
            then
                >$i
            else
                echo "file $i does not exist"
            fi
        ;;
    esac
done

```

The `-c` flag in this command forces subsequent files to be created if they do not already exist. Without the `-c` flag, an error message prints if the file does not exist. The shell variable `flag` is set to some non-null string if the `-c` argument is encountered.

The exit status of the `if` command is the exit status of the last command following a `then` or `else`. If no such commands are executed, then the exit status is zero.

Conditional execution of commands can also be achieved with the symbols `&&` and `||`. See “Conditional Execution” for details.

16.6 `exit`

A shell script terminates when it reaches `eof`. The exit status of the script is that of the last command executed. The built-in `exit` command can cause the script to terminate with exit status set to *n*. If *n* is omitted, exit status is that of the last command executed before `exit` was encountered.

17. Input and output

17.1 I/O redirection

All forms of input and output redirection are allowed in shell scripts. If input or output redirection (using `<` or `>`) is done in any of the control-flow commands, the entire command is executed in a subshell. This means that any values assigned during execution of the command will not be available after the command is over and control returns to the parent shell. If necessary, you can change the shell's standard input and output. See "Changing the Shell's Standard Input and Output."

17.1.1 Redirection with file descriptors

The A/UX system considers standard input, standard output, and standard error output as files, and associates a file descriptor with each of them.

File descriptors are numbers used to identify files. File descriptors run from 0 to `(OPEN_MAX-1)` (see `intro(2)` in *A/UX Programmer's Reference*). By default, the file descriptors 0, 1, and 2 have the following associations:

- 0 is associated with standard input.
- 1 is associated with standard output.
- 2 is associated with standard error output.

Thus, standard input can be referenced via file descriptor 0, standard output can be referenced via file descriptor 1, and standard error can be referenced via file descriptor 2.

Input and output redirection uses the syntax

```
[x]< filename
```

and

```
[x]> filename
```

where *x* is an optional file descriptor number indicating a file; `>` and `<` are redirection operators; and *filename* is a file containing input, or to which output will be directed. The simple forms omit the file descriptor *x* and use the defaults listed above. If no descriptor appears, it is assumed to be 0 for input redirection and 1 for output redirection.

Standard error output must be redirected explicitly using a numeric file descriptor as documented below. The >> form may be used to append output to an existing file rather than overwrite the file's contents.

All file descriptors can be used with redirection characters in a command line, immediately preceding the redirection symbol. For example,

```
cc x.c 2>&1 | more
```

redirects standard error on top of standard output and pipes the result through `more`. Note that there must be no spaces between the characters in “`2>&1`”.

In all forms, specifications are evaluated by the shell from left to right as they appear in the command. Filenames are subject to variable and command substitution only. No filename expansion or blank interpretation takes place; for example, the command

```
cat testfile > *.c
```

simply writes `testfile` into a file named “`*.c`”.

17.1.2 File descriptors redirecting input

The default file descriptor for redirecting standard input is 0. This may be specified as

```
cat 0<filename
```

Because this is the default file descriptor, it may be omitted as follows:

```
cat <filename
```

17.1.3 File descriptors redirecting output

The default file descriptor for redirecting output is 1. This may be specified as

```
cat 1>filename
```

Because this is the default file descriptor, it may be omitted as follows:

```
cat >filename
```

17.1.4 Combining standard error and standard output

The default file descriptor for redirecting standard error output is 2. If you want to direct the error output of a command to a file (to save the

error messages), use the syntax

```
ls filename 2>errors
```

This saves error output (for example, “*filename* not found”) in a file named `errors`. If you want to save the command output and error output in separate files, use the syntax

```
ls filename >output 2>errors
```

To print the output and the error output in the same file, use the syntax

```
ls filename >output 2>&1
```

This writes both standard output and error output in the file `output`. Note that `2>&1` references the `output` file because you have already redirected standard output (file descriptor 1) to this file.

For example, to save the output and the error output of the `make` command in a file named `make.log`, use the command

```
make > make.log 2>&1
```

17.1.5 Changing the shell’s standard input and output

To associate standard input or standard output with a file, use the `exec` command:

```
exec >filename (for standard output)  
exec <filename (for standard input)
```

Output will be written to, or input taken from, the file specified until further redirection is done with the `exec` command. This can be useful if all output is to be taken from a file or written to a file. This construct is unlike normal shell redirection with `>` and `<` in that the redirection remains in effect until you log out or explicitly reset the standard I/O files.

To return output and input to the terminal, use the commands

```
exec > /dev/tty (for output)  
exec < /dev/tty (for input)
```

Reassignment can be used to avoid the problems involved in redirecting output or input in a control-flow structure.

17.1.6 Associating file descriptors with other files

The `exec` command can also be used to associate file descriptors with specific files. This can be an advantage in shell scripts that need to read or write a file line by line (see also “Reading Input”), because writing output to a file descriptor cannot overwrite a file’s contents. The command syntax

```
exec x<filename
```

where *x* is a number [3 to (OPEN_MAX-1)], associates *filename* with *x* (see `intro(2)` in *A/UX Programmer’s Reference* for a definition of OPEN_MAX). For example, the commands

```
exec 4<file1
exec 5<file2
```

associate file descriptor 4 with *file1* and file descriptor 5 with *file2*. After these commands, the syntax

```
command <&4
```

takes input from *file1* and

```
command >&5
```

writes output to *file2*. Using the ampersand (&) prevents the shell from creating or looking for a file named 4 or 5 in these examples.

The following example shows how the `>&n` file descriptor syntax may be used:

```
$ exec 4>file2
$ echo hello >&4
$ cat file2
hello
$ echo bye >&4
$ cat file2
hello
bye
```

Note that this file descriptor syntax can be repeated in a loop without overwriting the contents of *file2*.

17.2 Reading input

The built-in `read` command reads a line of input from the terminal or a file and assigns it to the variables specified. The form of the `read` command is

```
read [name...]
```

One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. If only one *name* is specified, the entire line read will be assigned to that *name*. The exit status is zero while there is data to be read. If an *eof* or an *interrupt* is encountered, the exit status is nonzero.

For example, you could use the `read` command to take input from the terminal as follows:

```
$ read first middle last abbreviations
Alyssa Elizabeth Lynch Dr. Ph.D.
```

This would result in the following variables assignments:

```
first=Alyssa
middle=Elizabeth
last=Lynch
abbreviations=Dr. Ph.D.
```

The `read` command can also take input from a file, but will always read the first line. If you wish to move sequentially through a file, reading it line by line, you must first use the `exec` command to make the file standard input as follows:

```
exec < name.list
while read first middle last abbreviations
do
    command-list
done
exec < /dev/tty
```

In the above example, the `exec` command is used to reassign standard input to the file “`name.list`”. The `while` loop then uses the `read` command to read each line of the file into the variables `first`, `middle`, `last`, and `abbreviations`, and execute *command-list*.

When `read` reaches the end of the file, it will return a nonzero exit status and the `while` loop will terminate. The final `exec` command then assigns standard input back to the terminal. For information about reassignment with the `exec` command, see the preceding section.

The A/UX `line` command functions exactly like the `read` command, except that a whole line is read into a single variable. The line will be terminated with a newline.

17.3 Taking input from scripts

Input to a shell script can be embedded inside the script itself. This is called a **here document**. The information in a here document is enclosed as follows:

```
<<[-] word
    information
word
```

The first *word* may appear anywhere on a line; the second must appear alone on a line, that is, it cannot be indented. The *words* must be identical and should not be anything that will appear in *information*. The second *word* is the end-of-file for the here document. Variable and command substitution will occur on *information*. Normal quoting conventions apply, so that `$` can be escaped with `\`. To prevent all substitution, quote any character of the first instance of *word*. (If substitution is not required, this is more efficient.)

To strip leading tabs from *word* and *information*, precede the first instance of *word* with the optional hyphen (`-`), as follows:

```
<<-word
```

Note: If you intend to indent your code, you must use the hyphen preceding *word* unless the commands you use can tolerate leading tabs.

For example, a shell procedure could contain the lines

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

Here the `grep` command looks for the pattern specified by `$i` in the file `/usr/lib/telnetd`. This file could contain the lines

```
fred mh0123
bert mh0789
```

An alternative to using an external file would be to include this data within the shell procedure itself as a here document:

```
for i
do
    grep $i <<!
    ...
    fred mh0123
    bert mh0789
    ...
!
done
```

In this example, the shell takes the lines between `<<!` and `!` as the standard input for `grep`. The second `!` represents the *eof*. The choice of `!` is arbitrary. Any string can be used to open and close a here document, provided that the string is quoted if white space is present and the string does not appear in the text of the here document.

Here documents are often used to provide the text for commands to be given for interactive processes, such as an editor, called in the middle of a script. For example, suppose you have a script named `change` that changes a product name in every file in a directory to a new name, as follows:

```

for i in *
do
echo $i
ed $i <<!
g/oldproduct/s//newproduct/g
w
!
done

```

(Note that `ed` commands will not tolerate leading tab characters and there is no hyphen preceding the first *word*, therefore the code is not indented.) The metacharacter `*` is expanded to match all filenames in the current directory, so the `for` loop executes once for each file. For each file, the `ed` editor is invoked. The editor commands are given in the here document between `<<!` and `!`. They direct the editor to search globally for the string `oldproduct` and each time it is found substitute the string `newproduct`. After the substitution is made, the editor saves the new copy of the file with the `w` command.

You could make the `change` script more general by using parameter substitution, as follows:

```

for i in *
do
echo $i
ed $i <<!
g/$1/s//$2/g
w
!
done

```

Now the old and new product names (or any other strings) can be given as positional parameters on the command line:

```
change string1 string2
```

Substitution of individual characters can be prevented by using a backslash (`\`) to quote the special character `$`, as in

```

for i in *
do
echo $i
ed $i <<!
1, \${s}/${1}/${2}/g
w
!
done

```

This version of the script is equivalent to the first, except that the substitution is directed to take place on the first to the last lines of the file (1, \$) instead of “globally” (g) as in the first example. This way of giving the command has the advantage that the editor will print a question mark (?) if there are no occurrences of the string \$1.

Substitution can be prevented entirely by quoting the first instance of the terminating string; for example,

```
ed $i <<\

```

Note that backslash, single quotes and double quotes all have the same effect in this context: they turn off variable expansion and filename expansion.

To prevent leading tabs from becoming part of the here document, precede the first *word* with a hyphen, as follows:

```

for i in *
do
    echo $i
    ed $i <<-!
        1, \${s}/${1}/${2}/g
    w
!
done

```

17.4 Using command substitution

Command substitution can occur in all contexts where variable substitution occurs. You can use command substitution in a shell script to avoid typing long lists of filenames. For example,

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

runs the `ex` editor, supplying as arguments those files whose names end in `.c` and that contain the string `TRACE`. Another example,

```
for i in `ls -t`  
do  
    command-list $i  
done
```

sets the variable `i` to each consecutive filename in the current directory, with the most recent filename first.

Command substitution is also used to generate strings. For example,

```
set `date`; echo $6 $2 $3, $4
```

first sets the positional parameters to the output of the `date` command and then will print; for example,

```
1986 Nov 1, 23:59:59
```

Another common example of command substitution uses the `basename` command. `basename` removes the suffix from a string, so

```
basename main.c .c
```

prints the string `main`. The following fragment illustrates its application in a command substitution:

```
case $A in  
    ...  
    *.c)    B=`basename $A .c`  
    ...  
esac
```

Here `B` is set to the part of `$A` with the suffix `.c` stripped off.

17.5 Writing to standard output

The `echo` command is used to write to standard output (by default, the terminal). The form of the `echo` command is

```
echo arguments escapes
```

The *arguments* are what is written. They are evaluated like the

arguments of any other command with variable and command substitution, filename expansion, and blank interpretation. Normal quoting conventions apply. Strings containing blanks must be enclosed in double quotes. The arguments will be written sequentially separated by blanks, and by default they will be terminated with a newline. If there are no *arguments* or the *arguments* are unset or null variables, a blank line will be returned.

The *escapes* indicate how the *arguments* should be printed. The possible escapes are

- `\b` backspace
- `\c` print line without newline
- `\f` form feed
- `\n` newline
- `\r` carriage return
- `\t` tab
- `\v` vertical tab
- `\\` backslash
- `\n` the 8-bit character whose ASCII code is the 1-, 2-, or 3-digit octal number *n*, which must start with a zero

The backslash in each escape must be quoted; that is, it must appear twice or be enclosed in quotes. Escapes can occur anywhere among the arguments. For example, to produce two lines of output with a single `echo` command, you could give the command

```
echo "line one"\\n"line two"
```

To print the value of a variable and keep the cursor in the same line, you could give the command

```
echo $jj\\c
```

See `echo(1)` in *A/UX Command Reference* for more information.

18. Other features

18.1 Arithmetic and expressions

The Bourne shell has no built-in arithmetic. The A/UX `expr` command can be used for integer arithmetic, logical operations, comparison, and some pattern matching and creation of substrings.

Integers and operands are passed to the `expr` command as separate arguments, which means that they must be separated by spaces as follows:

```
expr 1 + 1
```

Shell metacharacters such as the asterisk (*), must be quoted with the backslash (\). For instance, to have the shell compute the value of 5 factorial (in symbols: 5!), you could enter

```
expr 5 \* 4 \* 3 \* 2
```

The following are some of the operators allowed in `expr` expressions, in increasing precedence:

= > >= < <= !=

These symbols return the result of an integer comparison if both arguments are integers; otherwise they return the result of a lexical comparison.

+ -

These symbols return the result of addition or subtraction of integer-valued arguments.

* / %

These symbols return the result of multiplication or division, or the remainder of the integer-valued arguments.

For a complete list, see `expr(1)` in *A/UX Command Reference*.

The primary use of `expr` is in command substitution to set variables. For example, to count the iterations of a loop, you could increment the variable `a` as follows:

```
a=`expr $a + 1 `
```

The `expr` command can also be used to pick apart strings and do pattern matching. To perform floating-point calculation, use `awk` or

bc. See *A/UX Programming Languages and Tools, Volume 2* for details.

18.2 File status and string comparison

The built-in `test` command evaluates an expression and returns a zero (true) exit status if the expression is true, and a nonzero (false) exit status if the expression is false or there is no argument. It is often used in the shell control-flow constructs.

For example,

```
test -f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. Some of the more frequently used `test` arguments are given below. See “Summary of Bourne Shell Commands” at the end of this chapter for a complete list.

Note: Because people often name test programs `test`, you may obtain unpredictable results using the `test` command as well. A harmless alternative is the `[]` construct, such as

```
if [ -f file ]
then
    command-list
fi
```

<code>test <i>s</i></code>	True if <i>s</i> is not the null string.
<code>test <i>s1</i> = <i>s2</i></code>	True if <i>s1</i> and <i>s2</i> are identical.
<code>test <i>s1</i> != <i>s2</i></code>	True if <i>s1</i> and <i>s2</i> are <i>not</i> identical.
<code>test -f <i>file</i></code>	True if <i>file</i> exists.
<code>test -r <i>file</i></code>	True if <i>file</i> exists and is readable.
<code>test -w <i>file</i></code>	True if <i>file</i> exists and is writable.
<code>test -d <i>file</i></code>	True if <i>file</i> exists and is a directory.

`test n1 -eq n2` True if the integers *n1* and *n2* are algebraically equal. Any of the comparisons `-ne`, `-gt`, `-ge`, `-lt`, and `-le` may be used in place of `-eq`.

In addition, there are the following operators:

`!` the unary negation operator

`-a` binary AND operator

`-o` binary OR operator

The `-a` operator has higher precedence than `-o`.

All the operators and flags are separate arguments to `test`.

Parentheses can be used for grouping, but must be escaped with the backslash.

The following is a typical use of the `test` command in a shell script:

```
if test -d foo
then
    echo "foo is a directory"
fi
```

This prints the message “foo is a directory” if `foo` is found to be a directory when the `test` command is run.

There is also an alternate name for the `test` command, the left bracket, `[`. When invoked under this name, the following form works identically to the example above:

```
if [ -d foo ]
then
    echo "foo is a directory"
fi
```

Be sure to surround each bracket with spaces, or they will not be recognized as a command.

18.3 The null command (:)

The null command `(:)` does nothing and returns a zero exit status. The form of the command is

: *args*

This command can also be used wherever `true` can be used; for example,

```
while : args
```

19. Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively.

Execution of a command may fail for any of the following reasons:

- I/O redirection may fail if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a bus error or memory fault signal.
- The command terminates normally but returns a nonzero exit status.

In most cases, the shell will print an error message and go on to execute the next command. An interactive shell will return to read another command from the terminal. If the command is a shell script, nonzero exit status or abnormal termination of a command may allow the script to continue on to execute the next command.

Other types of errors, such as failed I/O redirection, command not found, syntax errors such as “`if then done`”, an *interrupt* signal that was not trapped, or failure of any of the built-in commands usually cause a script to terminate.

The shell flag `-e` causes the shell to terminate if an error is detected.

19.1 Fault handling and interrupts

The A/UX system uses **signals** to communicate between processes. Most signals indicate an interrupt, termination, error condition, or other break in processing. See `signal(3)` in *A/UX Programmer's Reference* for more information.

The signals that are likely to be of interest in fault handling are

- 1, hangup
- 2, interrupt
- 3, quit
- 14, alarm clock
- 15, software termination (kill)

When a process receives a signal, it can handle it in one of three ways:

- Signals can be ignored. Some signals will cause a core dump if they are not caught.
- Signals can be caught, in which case the process must decide what action to take when the signal is received.
- Signals can be left to cause termination of the process without further action.

Note: The built-in `trap` command is suitable only for simple signal handling (for example, catching an *interrupt* from the keyboard in order to terminate the script). Functions requiring complex signal handling should be implemented as a C program. See *A/UX Programming Languages and Tools, Volume 1* for more information about the C programming language and associated library routines.

The built-in `trap` command allows you to detect error signals and indicate what action should be taken. The command has the form

```
trap [command] [number] ...
```

command is a command string that is read and executed when the shell receives signals whose numbers are given in *number*. *command* is scanned once when the trap is set and once when the trap is executed. `trap` commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error.

The `trap` command with *numbers* but without any arguments resets the corresponding signals to their original values. If *command* is the

null string, the signal whose *number* is given is ignored by the shell and by the commands it invokes. If *number* is 0, *commands* are executed on normal termination from the shell script. The `trap` command with no arguments prints a list of commands associated with each signal number.

For example,

```
trap 'rm -f /tmp/junk; exit' 0 1 2 3 15
```

sets a trap for the specified signals and if any one of these signals is received, it will execute the following commands:

```
rm -f /tmp/junk; exit
```

It removes the temporary file `/tmp/junk` and then exits from the script. (`exit` is a built-in command that terminates execution of a shell procedure.) The `exit` is required; otherwise after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

The use of `trap` is illustrated in the following script:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
  case $i in
    -c)  flag=N ;;
    *)  if test -f $i
        then
            ln $i junk$$; rm junk$$
        elif test $flag
        then
            echo "file '$i' does not exist"
        else
            >$i
        fi ;;
  esac
done
```

The cleanup action is to remove the file `junk$$`. (This file is named after the process ID of the script, which is kept in the

system-maintained variable `$`; see “Parameters and Variables Set by the Shell.”) 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.

A procedure may itself elect to ignore signals by specifying the null string as the argument to `trap`. The fragment

```
trap '' 1 2 3 15
```

causes the system hangup, interrupt, quit, and software termination signals to be ignored both by the procedure and by invoked commands. These settings could be listed with the `trap` command without arguments, and reset by entering

```
trap 1 2 3 15
```

which resets the traps for the corresponding signals to their default values.

The following `scan` procedure is an example of using `trap` where there is no exit in the `trap` command:

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

This procedure steps through each directory in the current directory, prompts with its name, and then executes commands entered at the terminal until an *eof* or an *interrupt* is received. Interrupts are ignored while executing the requested commands but cause termination when `scan` is waiting for input.

19.2 Debugging a shell script

Several shell options can be set that will help with debugging shell scripts. These are

- e e (error) causes the shell to exit immediately if any command exits with a nonzero exit status. (This can be dangerous in scripts involving `until` loops and other constructs where nonzero exit status is desired.)
- n n (no execute) prevents execution of subsequent commands. Commands will be evaluated but not executed. This is usually combined with the `-v` option when used for debugging. (Note that typing `set -n` at a terminal will render the terminal useless until an *eof* is entered.)
- u u (unset) causes the shell to treat unset variables as an error condition.
- v v (verbose) causes lines of the procedure to be printed as read. Use this to help isolate syntax errors.
- x x provides an execution trace. Following parameter substitution, each command is printed as it is executed.

These execution options can be turned on with the `set` command:

```
set -option
```

either inside the script or before its execution (except `-n`, which will freeze the terminal until you send an *eof*). Options can be turned off by typing

```
set +option
```

Alternatively, they can be turned on with the `sh` command if the script is executed this way. The current setting of the shell flags is available as `$-`.

20. Summary of Bourne shell commands

Input/output redirection is permitted for these commands. File descriptor 1 is the default output location.

- : No effect; the command does nothing. A zero exit code is returned. See “The Null Command (:).”

- `. file`
Read and execute commands from *file* and return. The search path specified by `PATH` is used to find the directory containing *file*. Note that the dot command does not spawn a subshell. See “Executing Shell Scripts.”
- `break [n]`
Exit from the enclosing `for` or `while` loop, if any. If *n* is specified, break *n* levels. See “Control-Flow Constructs.”
- `cd [arg]`
Change the current directory to *arg*. The environment variable `HOME` is the default *arg*. The environment variable `CDPATH` defines the search path for the directory containing *arg*. If *arg* begins with `/`, the search path is not used. Otherwise each directory in the path is searched for *arg*. See “The Environment.”
- `continue [n]`
Resume the next iteration of the enclosing `for` or `while` loop. If *n* is specified, resume at the *n*th enclosing loop. See “Control-Flow Constructs.”
- `eval [arg ...]`
Read arguments as input to the shell and execute the resulting commands. See “Forcing More Than One Pass of Evaluation.”
- `exec [arg ...]`
Execute the command specified by the arguments in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified. See “Command Execution.”
- `exit [n]`
Cause the shell to exit with the exit status specified by *n*. If *n* is omitted, the exit status is that of the last command executed. (An *eof* will also cause the shell to exit.) See “Working With More Than One Shell.”
- `export [name ...]`
Mark *names* for automatic export to the environment of subsequently executed commands. If no arguments are given, a

list is printed of all names exported in the current shell. Function names may *not* be exported. See “The Environment.”

hash [-r] [name ...]

For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The -r option causes the shell to forget all locations. If no arguments are given, *hits* and *cost* about remembered commands are presented. *hits* is the number of times a command has been invoked by the shell process. *cost* is a measure of the work required to locate a command in the search path. There are certain situations that require that the stored location of a command be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *cost* will be incremented when the recalculation is done. See “Writing Efficient Shell Scripts.”

newgrp [arg ...]

Equivalent to “exec newgrp arg ...”. This built-in version executes faster than the A/UX command but is otherwise identical. See newgrp(1) in *A/UX Command Reference* for usage and description.

pwd

Print the current working directory. This built-in version executes faster than the A/UX command but is otherwise identical. See pwd(1) in *A/UX Command Reference* for usage and description.

read [name ...]

Read one line from the standard input and assign the first word to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. The exit status is 0 unless an *eof* is encountered. See “Writing Interactive Shell Scripts.”

readonly [name ...]

Mark *names* read-only. The values of these *names* cannot be changed by subsequent assignment. If no arguments are given, a list of all read-only names is printed. See “Setting Constants.”

`return [n]`

Cause a function to exit with the return value specified by *n*. If *n* is omitted, the exit status is that of the last command executed. See “Defining Functions.”

`set [[-][-aefhkntuvx][arg ...]]`

- a Mark variables that are modified or created for export.
- e Exit immediately if a command terminates with a nonzero exit status.
- f Disable filename expansion.
- h Locate and remember function commands as functions that are defined (function commands are normally located when the function is executed).
- k Place all keyword arguments in the environment for a command, not just those that precede the command name.
- n Read commands but do not execute them.
- t Exit after reading and executing one command.
- u Treat unset variables as an error when substituting.
- v Print shell input lines as they are read.
- x Print commands and their arguments as they are executed.
- Do not change any of the flags; useful in setting \$1 to -.

Using + rather than - causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in \$-. The remaining arguments are positional parameters and are assigned, in order, to \$1, \$2, and so on. If no arguments are given, the values of all names are printed. See “The Environment” and “Shell Execution Options.”

`shift [n]`

Change the names of the positional parameters \$*n*+1 ... to \$1 ... If *n* is not given, it is assumed to be 1. See “Changing Parameter Positions.”

`test [expr]`

Evaluate conditional expressions. `test` evaluates the expression *expr* and, if its value is true, returns a zero (true) exit status; otherwise, a nonzero (false) exit status is returned. `test` also returns a nonzero exit status if there are no arguments. The superuser is always granted execute permission even though (1) execute permission is meaningful only for directories and regular files, and (2) `exec` requires that at least one execute mode bit be set for a regular file to be executable.

The following primitives are used to construct *expr*:

- `-r file` True if *file* exists and is readable.
- `-w file` True if *file* exists and is writable.
- `-x file` True if *file* exists and is executable.
- `-f file` True if *file* exists and is a regular file.
- `-d file` True if *file* exists and is a directory.
- `-c file` True if *file* exists and is a character special file.
- `-b file` True if *file* exists and is a block special file.
- `-p file` True if *file* exists and is a named pipe (FIFO).
- `-u file` True if *file* exists and its set user ID bit is set.
- `-g file` True if *file* exists and its set group ID bit is set.
- `-k file` True if *file* exists and its sticky bit is set.
- `-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.
- `-z s1` True if the length of string *s1* is zero.
- `-n s1` True if the length of the string *s1* is nonzero.
- `s1 = s2` True if strings *s1* and *s2* are identical.

s1 != s2 True if strings *s1* and *s2* are *not* identical.

s1 True if *s1* is *not* the null string.

n1 -eq n2 True if the integers *n1* and *n2* are algebraically equal. Any of the comparisons *-ne*, *-gt*, *-ge*, *-lt*, and *-le* may be used in place of *-eq*.

These primaries may be combined with the following operators:

! unary negation operator

-a binary AND operator

-o binary OR operator (-a has higher precedence than -o)

(*expr*) parentheses for grouping

Notice that all the operators and flags are separate arguments to `test`. Notice also that parentheses are meaningful to the shell and, therefore, must be escaped.

`test` is typically used in shell scripts, as in the following example, which prints the message “foo is a directory” if it is found to be one when `test` is run:

```
if test -d foo
then
    echo "foo is a dir"
fi
```

`times`

Print the accumulated user and system times for processes run from the shell. See “Writing Efficient Shell Scripts.”

`trap [arg] [n] ...`

Read the command *arg* and execute when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) `trap` commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is absent, all trap(s) *n* are reset to their original values. If *arg*

is the null string, this signal is ignored by the shell and by the commands it invokes. If *n* is 0, the command *arg* is executed on exit from the script. See “Fault Handling and Interrupts.”

`umask nnn`

Set the file-creation mask to *nnn*. The three octal digits refer to read/write/execute permissions for *owner*, *group*, and *others* respectively (see *chmod(2)* and *umask(2)*). The value of each specified digit is subtracted from the corresponding “digit” specified by the system for the creation of a file (see *creat(2)*). For example, `umask 022` removes *group* and *others* write permission (files normally created with mode 777 become mode 755; files created with mode 666 become mode 644). If the argument *nnn* is omitted, the current value of the mask is printed.

Chapter 3

Korn Shell Reference

Contents

1. The Korn shell prompt	1
1.1 The secondary shell prompt	1
1.2 The tertiary shell prompt	1
1.3 Changing the prompt character	1
2. Types of commands	2
2.1 Learning about built-in commands	2
3. The parts of a command	3
4. Interactive use	4
4.1 Command termination character	4
4.2 Impossible commands	4
4.3 Background commands	4
4.3.1 Checking command status	5
4.3.2 Logging out	5
4.4 Canceling commands	5
4.4.1 Before you press RETURN	6
4.4.2 While a command is running	6
4.4.3 Canceling background commands	7
5. Editing and reusing commands	8
5.1 The <code>vi</code> option	9
5.1.1 The editor window	9
5.1.2 Command history	10
5.1.3 Moving the cursor on the command line	11
5.1.4 Changing and inserting text in the command line	12
5.1.5 Deleting text from the command line	13

5.1.6	Copying and moving text within the command line	13
5.1.7	Specialized editing commands	13
5.1.8	Printing and executing edited commands	14
5.2	The emacs (and gmacs) options	14
5.2.1	The emacs input edit commands	15
5.2.2	The emacs cursor motion commands	15
5.2.3	The emacs history commands	15
5.2.4	The emacs text modification commands	16
5.2.5	Other emacs editing commands	17
5.3	Using fc or r	18
5.3.1	Editing and reexecuting previous commands	18
5.3.2	Listing previous commands	20
6.	Using shell metacharacters	21
6.1	Shortcuts in working with directories	23
6.1.1	Specifying home directories	23
6.1.2	Current and previous directories	23
6.1.3	Substituting directory names	24
6.2	Specifying filenames with metacharacters	24
6.3	Input and output redirection	27
6.4	Combining commands: pipelines	28
6.5	Connecting a command to standard input and output	29
6.6	Command grouping	30
6.7	Conditional execution	31
6.8	Quoting	32
7.	Working with more than one shell	33
7.1	Changing to a new shell	34
7.2	Changing your default shell	34
8.	The environment	34
8.1	Listing existing values	35
8.2	Assigning values to environment variables	36

8.3	Removing environment variables	36
8.4	Commonly used environment variables	36
8.5	The environment and new shell instances	39
8.6	Special environments	40
8.7	The default environment on your system	41
9.	The <code>.profile</code> file	42
9.1	A sample <code>.profile</code> file	43
9.1.1	Locating commands	43
9.1.2	Shortcuts in changing directories	44
9.1.3	Receiving mail	45
9.1.4	Your editing environment	45
9.2	Customizing your login procedure	45
10.	The <code>.kshrc</code> file	46
10.1	A sample <code>.kshrc</code> file	46
10.1.1	Changing history variables	46
10.2	Changing the <code>ENV</code> filename	47
11.	Aliases for commonly used commands	47
11.1	Defining an alias	48
11.2	Listing and removing aliases	49
11.3	Tracking with aliases	49
11.4	Default aliases	50
12.	Shell execution options	50
12.1	Options that affect the environment	51
12.2	Options for invoking new shells	51
13.	Job control	52
13.1	Suspending a job	52
13.2	Listing jobs	53
13.3	Changing the status of stopped jobs	53
13.4	Blocked jobs	55
13.5	Canceling jobs	55
13.6	Logging out with stopped jobs	56
14.	Using shell layering	56
15.	Overview of shell programming	56
15.1	Writing shell programs	57
15.2	Executing shell scripts	58

15.3	Comments	59
15.4	Writing interactive shell scripts	59
15.5	Canceling a shell script	60
15.6	Writing efficient shell scripts	60
16.	Command evaluation	61
16.1	Forcing more than one pass of evaluation	63
16.2	Command execution	63
16.3	Exit status: the value of the command	64
17.	Defining functions	64
18.	Positional parameters and shell variables	66
18.1	Positional parameters	67
18.1.1	Setting values in a script	67
18.1.2	Changing parameter positions	68
18.1.3	Number of parameters	69
18.2	Shell variables	69
18.2.1	Assigning values	69
18.2.2	Arrays of strings	70
18.2.3	Assigning values and types to variables	70
18.2.4	Assigning values on the command line	76
18.2.5	Removing shell variables	76
18.3	Setting constants	76
18.4	Parameter and variable substitution	77
18.5	Referencing arrays	78
18.6	Testing assignment and setting defaults	79
18.7	Creating substrings in substitution	80
18.8	Parameters and variables set by the system	81
19.	Control-flow constructs	82
19.1	for loops	83
19.2	select statements	84
19.3	case statements	85
19.4	while loops	87
19.5	until loops	88
19.6	if then else	89
19.7	exit	91

20. Input and output	92
20.1 I/O redirection	92
20.1.1 Redirection with file descriptors	92
20.1.2 File descriptors redirecting input	93
20.1.3 File descriptors redirecting output	93
20.1.4 Combining standard error and standard output	93
20.1.5 Changing the shell's standard input and output	94
20.1.6 Associating other files with file descriptors	95
20.2 Reading input	95
20.3 Taking input from scripts	97
20.4 Using command substitution	100
20.5 Writing to the standard output	101
20.6 Creating and reading a menu	103
21. Other features	105
21.1 Arithmetic evaluation	105
21.2 File status and string comparison	107
21.3 The null command (:)	109
22. Error handling	109
22.1 Fault handling and interrupts	110
22.2 Debugging a shell script	113
23. Summary of Korn shell commands	114

Tables

Table 3-1. Listing functions, aliases, and variables	35
---	-----------

Chapter 3

Korn Shell Reference

1. The Korn shell prompt

The Korn shell is a program that interprets commands and arranges for their execution. The Korn shell displays a character called the **prompt** (or **primary shell prompt**) whenever it is ready to begin reading a new command from the terminal. By default, the Korn shell prompt character is set to the dollar sign (\$).

1.1 The secondary shell prompt

If you press the RETURN key when the shell expects further input, you will see the **secondary shell prompt**. By default, this prompt character is set to the greater-than sign (>). Like the primary shell prompt, this can be redefined.

The secondary prompt will appear, for example, if you enter a multiline construct (such as a function definition) at the primary shell prompt. The secondary prompt will appear at each line until you give the final delimiter. Whenever you have a secondary prompt (either because you are using a multiline construct or because of an error), an *interrupt* will stop the process and issue a primary prompt (\$) for another command. See “Canceling Commands” for information about the *interrupt* on your system.

1.2 The tertiary shell prompt

If you use the `select` command to set up a menu, the **tertiary shell prompt** displays on lines that prompt for a user selection. By default, the tertiary shell prompt is set to “#?”.

1.3 Changing the prompt character

You may change the primary prompt character by redefining the environment variable `PS1` to any other character or string of characters. Similarly, the secondary shell prompt can be redefined, if desired, by changing the environmental variable `PS2`, and the tertiary prompt, by changing the setting of `PS3`. See “Commonly Used Environment Variables.”

2. Types of commands

The shell works with three types of commands:

Built-in shell commands: Built-in commands are written into the shell itself and are generally used for writing shell programs. Each A/UX shell has a slightly different set of built-in commands. The built-in Korn shell commands are listed under “Summary of Korn Shell Commands.”

A/UX commands: Every shell can invoke all A/UX commands (see “Command Summary by Function” in *A/UX Command Reference* for a complete list of these). A/UX commands are executable programs stored in system directories such as `/bin` and `/usr/bin`. When you enter an A/UX command (for example, `ls`), the shell searches all directories specified by your `PATH` variable (see “Locating Commands”) to locate the program and invoke it.

User-defined commands: You can combine built-in shell commands and A/UX commands to define your own **shell programs** (see “Overview of Shell Programming”). Shell programs can be typed in at the shell prompt or entered in a file. A shell program contained in a file is generally called a **shell script**. Once a shell script is defined, with certain limitations, it can be used like any other command or program.

You can also write your own commands in a high-level language such as C (see *A/UX Programming Languages and Tools, Volume 1* for more information.) The names of user-defined commands should not be the same as any existing shell or A/UX command.

2.1 Learning about built-in commands

To learn about any Korn shell built-in command, use the `whence` command:

```
whence [-v] built-in
```

For example,

```
whence r
```

tells you about the Korn shell `r` command. It prints

```
fc -e -
```

Use the `-v` option for a more verbose report. For example,

```
whence -v r
```

prints

```
r is an exported alias for fc -e -
```

In addition, the full pathnames of commands are given. For example,

```
whence more
```

prints

```
/bin/more
```

3. The parts of a command

Whenever you see a shell prompt, you can enter a command by typing the command name. Most A/UX commands have one or more **flag options**, which follow the command name to modify the way the command operates. These are usually composed of a hyphen followed by one or more characters; for example, `-l` modifies the `ls` command:

```
ls -l
```

In this case, the `-l` changes the way the `ls` command operates, producing a “long” listing that contains more information than the standard `ls` output. For the options that apply to a particular A/UX command, see the manual page entry for that command in *A/UX Command Reference*. For options to the Korn shell built-in commands, see “Summary of Korn Shell Commands.”

Many A/UX commands also expect one or more **arguments**, which pass information to the command. An argument may be any data expected by the command; for example, a directory name may follow the `ls` command:

```
ls /bin
```

In this case, the directory name `/bin` specifies which directory the `ls` command should list.

The entire command name, including any options and arguments, is called the **command line**. A command line is terminated by RETURN. For example, in the command line

```
ls -l /bin
```

`ls` is the command name, `-l` is a flag option (specifying a “long” listing), and `/bin` is an argument (specifying which directory to list).

To give a command longer than one line, you must precede RETURN with a backslash (`\`). This prevents the shell from interpreting RETURN as the end of a command. You can continue this for several lines; the shell will wait for a plain RETURN (not preceded by a backslash) to execute the multiline command.

Commands can also be combined; see “Command Grouping.”

4. Interactive use

4.1 Command termination character

When you are entering commands interactively, the shell will not begin executing a command until you press the RETURN key. Therefore, if you mistype something, you can back up and correct the mistake before pressing RETURN. When the shell recognizes the RETURN, it executes the command line; when the process completes, a new prompt will be printed on the screen. The shell is now ready to accept further commands.

4.2 Impossible commands

If you give an impossible command (a command or command line that doesn’t exist or uses improper syntax), the shell will print an error message and return the prompt for another command.

4.3 Background commands

You can direct the shell to execute commands in the “background” while you continue to work at the shell prompt (the “foreground”). To run background processes, end the command line with an ampersand (`&`) before the final RETURN. For example,

```
cat smallfile1 smallfile2 > bigfile &  
[1] 1234
```

The number in `[]` is the **job number** (for job control). The other number is the **process ID (PID)** associated with the sample `cat` command as long as it is executing. After the process ID is displayed, the shell returns the prompt so you can use the terminal immediately for other work.

Note: To save the output from the job you are running in the background, you must redirect it into a file or pipe it to a printer. If you do not redirect the output, any output produced by the command will appear on your screen and will not be saved.

To suspend processes that require input from the keyboard (such as an editor or a remote login across a network), use shell layering (see “Using Shell Layering”) or job control (see “Job Control”).

4.3.1 Checking command status

To check on the status of a background command, use

```
ps
```

This command shows the **process status** of all your commands; they are identified by process number and by name. See `ps(1)` in *A/UX Command Reference* for details.

You can use the built-in command `jobs` to get the status of your current jobs.

4.3.2 Logging out

The shell terminates all processes when you log out of the system. To make sure that a process will continue to execute after you log out, use the `nohup` command (which stands for “no hang up”) as follows:

```
nohup command &
```

See `nohup(1)` in *A/UX Command Reference* for details.

`nohup` is on by default for background processes on the Macintosh II; other machines should use the command form above.

4.4 Canceling commands

A number of special control sequences come into play when canceling commands. The A/UX standard distribution defines these sequences as follows:

Name	A/UX standard distribution
<i>interrupt</i>	CONTROL-c
<i>quit</i>	CONTROL-l
<i>erase</i>	DELETE
<i>kill</i>	CONTROL-u
<i>eof</i>	CONTROL-d
<i>swtch</i>	CONTROL- `
<i>susp</i>	CONTROL-z

You may reassign any of these sequences, however, using the `stty` command. See `stty(1)` in *A/UX Command Reference* for more information.

4.4.1 Before you press RETURN

If you type part of a command and then decide you do not want to execute it, you can send an *interrupt* or *kill* to the system at any point in the command line.

4.4.2 While a command is running

There are several ways to stop a command that is executing:

- Send the *interrupt* signal.

For example, the output of a command such as

```
cat /etc/termcap
```

will scroll by on your terminal. If you want to terminate the process, you can send the *interrupt* signal. Because the `cat` command does not take any precautions to avoid or otherwise handle this signal, the *interrupt* will (eventually) cause it to terminate.

- Use CONTROL-s to suspend scrolling output.

The A/UX control-flow keys are CONTROL-s (suspend scrolling output) and CONTROL-q (resume scrolling output). You can use these to stop a screenful of output, resume scrolling, and stop a screenful again. CONTROL-s and CONTROL-q cannot be redefined using `stty`; however, `stty` can enable and disable control-flow.

- Send an *eof* character.

Many programs (including the shell) terminate when they get an *eof* character from their standard input. You could accidentally terminate the shell (which would log you off the system) if you enter *eof* at a prompt or, in terminating some other program, if you send an *eof* one time too many.

- Wait for the *eof* condition from a file.

If a command has its standard input redirected from a file, then it will terminate normally when it reaches the end of that file. If you give the command

```
mail ellen < note
```

(where `note` is an existing file), the `mail` program will terminate when it detects the *eof* condition from the file.

- Send the *quit* signal.

If you run programs that are not fully debugged, it may be necessary to stop them abruptly. You can stop programs that hang or repeat inappropriately using *quit*. This will usually produce a message such as

```
Quit (Core dumped)
```

indicating that a file named `core` has been created containing information about the state of the running program when it terminated because of the *quit* signal. You can examine this file yourself, or forward information to the person who maintains the program telling him or her where the `core` file is.

- Send a *suspend* signal.

If you are using shell layering, you can type `suspend` to stop jobs temporarily that are running on a shell layer. You can then resume the job with a special `shl` command. See Chapter 5, “Shell Layering.”

4.4.3 Canceling background commands

If you have a job running in the background and decide you do not want the command to finish executing, use the `kill` command.

When a job is running in the background, it ignores *interrupt* and *break* signals. To terminate a background command, use

```
kill process-ID
```

The `kill` command takes the process ID as an argument. See `kill(1)` and `ps(1)` in *A/UX Command Reference* for details.

You can also kill by job number, as in the C shell. For example,

```
kill %1
```

kills your first job.

5. Editing and reusing commands

The Korn shell provides access to an inline editor to edit your current command line or to edit past commands for reexecution. The inline editor option may be set at the shell prompt using the command

```
set -o option-name
```

where *option-name* may be

`vi` This option provides a window for the current command line and editing syntax similar to `vi`.

`emacs` or `gmacs`

Either of these options provide a window for the current command line and editing syntax similar to the `emacs` editor.

The only difference between the `emacs` and `gmacs` inline editors is the way they handle CONTROL-t.

If you set the value of the `EDITOR` environment variable to `vi`, `emacs`, or `gmacs`, the name of the inline editor will be taken from the environment automatically. See “The Environment” for more information.

Once you have supplied one of the above option names, you can invoke the inline editor on your current command line by pressing ESCAPE. The `vi` and `emacs` inline editors each have their own way of accessing your previous commands from a file named `$HOME/.sh_history`.

The Korn shell automatically saves the text of your past commands in the `$HOME/.sh_history` file, which is not an ordinary text file but

a special data file that can be read very quickly by the shell. Its contents are not lost when you log out. You can specify a special name for the history file with the environment variable `HISTFILE`, and the number of past commands you wish to access in the history file with the environment variable `HISTSIZE`. See “Commonly Used Environment Variables.”

Alternatively, you can use the `fc` command to access past commands and perform substitutions on them:

```
“fc -e -” or r
```

The `fc` command with the `-e` flag is aliased to `r`. You can use this command to perform substitutions on previous commands.

5.1 The `vi` option

Invoke the `vi` inline editor by pressing `ESCAPE`. If you have already started to enter a command when you press `ESCAPE`, the command will be displayed and the cursor will be on the last character you entered.

To exit the inline command editor and return to the shell prompt, press `CONTROL-d`. This will cancel the current command (the command in the editor window).

5.1.1 The editor window

While you are using the `vi` inline editor, your command line becomes a one-line editing screen. All of the `vi` commands listed below are available to you for editing commands, searching your command history, moving the cursor, and so on. There are several additional commands (not available in the full-screen `vi` editor) to perform filename generation, append arguments to previous commands, and so on.

The width of the screen will be 80 characters, unless you have set the `COLS` environment variable to some other width (see “Commonly Used Environment Variables”).

Command-line editing can be illustrated as follows:

1. Type in the command

```
cat defs chap.1 | troff -Tpsc -mm > L.2
```

2. Now you realize that you typed the wrong filename; it should be `chap.2`.
3. Press `ESCAPE`; then, using normal editing commands, move the cursor to the 1. (The quickest way to do this is by typing `f1`.) Now change the 1 to a 2. (The quickest way to do this is by typing `r2`.)
4. Now press `RETURN`. The command will execute as desired.

If the command is too long to fit in the window, the window will scroll with the cursor so that you can reach either end of the command. You will see a greater-than sign (`>`) on the right end of the command and a less-than sign (`<`) on the left end of the command. If both ends of the command are out of the window, you will see an asterisk (`*`).

5.1.2 Command history

The following commands give you access to your command history from command-line editing mode. Most take place as soon as they are typed; the search commands terminate with `RETURN`.

Note that the following commands may be preceded by a number to indicate how many times the command should execute (that is, if preceded by a number *n*, the command will execute the *n*th previous command, and so on).

- `k` Recall and print the most recent command. Each time `k` is entered, an earlier command is recalled. If preceded by a number *n*, the *n*th previous command is printed.
- `-` Equivalent to `k`.
- `j` Recall and print the next command in your history. Each time `j` is entered, a later command is recalled. If preceded by a number *n*, the *n*th next command is printed.
- `+` Equivalent to `j`.
- `[n]G` Recall command number *n*. If you don't supply the `G` command with a command number *n*, it defaults to a command number of one (1).
- `_` (Underscore) Insert the last argument of the most recent command to the current command and enter insert mode.

<i>/string</i>	Search backward in the history file for a previous command containing <i>string</i> and, if found, print it. <i>string</i> is terminated by RETURN. If <i>string</i> is null, the preceding string will be used.
? <i>string</i>	Search forward in the history file for the next command containing <i>string</i> and, if found, print it. <i>string</i> is terminated by RETURN. If <i>string</i> is null, the preceding string will be used.
<i>n</i>	Search for the next occurrence of the last <i>string</i> searched for with / or ?.
<i>N</i>	Search for the most recent occurrence of the last <i>string</i> searched for with / or ?.

5.1.3 Moving the cursor on the command line

These commands move the cursor around the current command line (the command line in the editor window). They take effect as soon as you enter them.

Note that the arrow keys *cannot be used* to move the cursor during inline editing.

The following commands may be preceded by a number to indicate how many times the command should execute (that is, if preceded by a number *n*, the command will move *n* spaces, *n* words, *n* lines, and so on, in that direction).

<i>h</i>	Move the cursor backward (left) one character.
<i>l</i>	Move the cursor forward (right) one character.
<i>w</i>	Move the cursor forward one alphanumeric word.
<i>W</i>	Move the cursor to the beginning of the next word that follows a blank.
<i>e</i>	Move the cursor to the end of the current word.
<i>E</i>	Move the cursor to the end of the word (ignoring quotes and other punctuating characters).
<i>b</i>	Move the cursor backward one word.

- B Move the cursor to the preceding word (ignoring quotes and other punctuating characters).
- 0 Move the cursor to the start of the line. (This cannot be preceded by *n*.)
- ^ Move the cursor to the first nonblank character in the line. (This cannot be preceded by *n*.)
- \$ Move the cursor to end of the line. (This cannot be preceded by *n*.)
- f*c* Search to the right for the next character *c* in the current line.
- F*c* Search to the left for the next character *c* in the current line.
- ;
, Repeat the last single character find (f or F) command.
Reverse the last single character find (f or F) command.

5.1.4 Changing and inserting text in the command line

These commands are used to replace characters in the current line and to add characters. Once the command is given, you can simply start typing the text you want. End the text you type with ESCAPE.

- a Append text after the cursor.
- A Append text after the end of the line.
- i Insert text before the cursor.
- I Insert text before the beginning of the line.
- cmotion* Change text. This command deletes from the current character through the character specified by the *motion* command (see the preceding section) and inserts the new characters typed. If *n* is included (preceding the *c* command or the *motion* command), the deletion covers the number of *motions* indicated.
- cc Change the entire line. If *n* follows this command, then *n* lines are discarded.

C Delete from the cursor to the end of the line and replace with the characters typed.

rc Replace the current character with *c*.

5.1.5 Deleting text from the command line

These commands are used to delete characters in the current command line. These commands take place as soon as they are typed.

D Delete from the cursor through the end of the line.

d*motion* Delete the current character through the character indicated by *motion*. If *n* is included (preceding the **d** command or the *motion* command), the deletion covers the number of *motions* indicated.

dd Delete the entire line. If *n* follows this command, then the deletion should cover the number of lines indicated.

x Delete the current character. If preceded by *n*, *n* characters are deleted.

5.1.6 Copying and moving text within the command line

P Place the last text modified before the cursor.

p Place the last text modified after the cursor.

5.1.7 Specialized editing commands

These commands take place as soon as they are entered.

. Repeat the most recent text modification command. If preceded by *n*, repeat the *n*th previous command that modified text.

~ Invert the case of the current character and advance the cursor.

u Undo the last text-modifying command.

U Undo all the text-modifying commands performed on the line.

***** Append an ***** to the current word and attempt filename generation. If no match is found, the bell rings. Otherwise the word is replaced by the matching pattern and insert

mode is entered.

5.1.8 Printing and executing edited commands

These commands take place as soon as you enter them. After they execute, you are returned to the Korn shell prompt.

CONTROL-I	(ell, not one) (form feed) Line feed and print the current line. This takes effect only when you are <i>not</i> entering text.
RETURN	Execute the current command line.
CONTROL-j	(line feed) Execute the current command line.
CONTROL-m	(RETURN) Execute the current command line.
#	Insert the character # as the first character in the command line. The # is the comment character, and everything after it will be ignored. This is useful for inserting the current line in history without being executed (although you will have to delete the initial # to reuse the command). This takes effect only when you are <i>not</i> entering text (that is, after you have pressed ESCAPE).

5.2 The emacs (and gmacs) options

The only difference between the `emacs` and the `gmacs` modes is the way they handle CONTROL-t. After you have enabled `emacs` mode (using “`set -o emacs`” or setting the value of the `EDITOR` variable), you can enter the `emacs` inline editor by pressing ESCAPE. You can then move the cursor to the point needing correction in your current command line and insert or delete characters or words as needed. All the editing commands are control characters or escape sequences. The notation for control characters is CONTROL-*letter*, where *letter* is a single (lowercase) character.

The notation for escape sequences is *M-* followed by a character. For example, *M-f* (pronounced “Meta f”) is entered by pressing ESCAPE (ASCII 033) and then pressing “f”. (*M-F* would be the notation for ESCAPE followed by “SHIFT” (uppercase) “F”.)

All edit commands operate from any place on the line (not just at the beginning). You do not press RETURN after editing commands except

where noted.

5.2.1 The `emacs` input edit commands

By default, the `emacs` editor is in input mode.

`erase` The *erase* character (see `stty(1)`). Delete previous character.

`eof` The *eof* character (see `stty(1)`). Terminate the shell if the current line is null.

`\` Escape next character. Editing characters and the *erase*, *kill*, and *interrupt* characters may be entered in a command line or in a search string if preceded by a `\`. The `\` removes the next character's editing features (if any).

`CONTROL-v` Display version of the shell.

5.2.2 The `emacs` cursor motion commands

The following commands move the cursor:

`CONTROL-f` Move the cursor forward (right) one character.

`M-f` Move the cursor forward one word. (A **word** is a string of characters consisting of only letters, digits, and underscores.)

`CONTROL-b` Move the cursor backward (left) one character.

`M-b` Move the cursor backward one word.

`CONTROL-a` Move the cursor to the start of the line.

`CONTROL-e` Move the cursor to the end of the line.

`CONTROL-]` *char*
 Move the cursor to character *char* on the current line.

`CONTROL-xCONTROL-x`
 Interchange the cursor and mark.

5.2.3 The `emacs` history commands

These commands access your command history:

`CONTROL-p` Fetch the previous command. Each time `CONTROL-p` is entered, the previous command back in time is accessed.

- M-<* Fetch the least recent (oldest) history line.
- M->* Fetch the most recent (youngest) history line.
- CONTROL-n Fetch the next command. Each time CONTROL-n is entered, the next command forward in time is accessed.
- CONTROL-r *string*
 Search backward in the history file for a previous command line containing *string*. If a parameter of zero is given, the search is forward. *string* is terminated by a RETURN or newline character. If *string* is omitted, then the next command line containing the most recent *string* is accessed. In this case, a parameter of zero reverses the direction of the search.
- CONTROL-o Execute the current line and fetch the next line relative to the current line from the history file.
- M-letter* Search the alias list for an alias by the name “*_letter*”, and if an alias of this name is defined, insert its value on the input queue. *letter* may not be one of the above metafunctions.
- M-.* Insert the last word of the previous command on the line. If preceded by a numeric parameter, the value of this parameter determines which word to insert rather than the last word.
- M-_
 Same as “M-.”.*
- M-** Attempt filename generation on the current word. An asterisk is appended if the word does not contain any special pattern characters.
- M-ESCAPE* Same as *M-**.
- M-=* List files matching current word pattern if an asterisk was appended.

5.2.4 The **emacs** text modification commands

These commands modify the line:

- CONTROL-d Delete the current character.

<i>M-d</i>	Delete the current word.
<i>M-CONTROL-h</i>	(Meta-backspace) Delete the previous word.
<i>M-h</i>	Delete the previous word.
<i>M-interrupt</i>	(Meta-interrupt) Delete the previous word. Note that if your <i>interrupt</i> character is DELETE, this command will not work.
CONTROL-t	Transpose the current character with the next character in <i>emacs</i> mode. Transpose two previous characters in <i>gmacs</i> mode.
CONTROL-c	Capitalize the current character.
<i>M-c</i>	Capitalize the current word.
<i>M-l</i>	Change the current word to lowercase.
CONTROL-k	Kill from the cursor to the end of the line. If given a parameter of zero, kill from the start of line to the cursor.
CONTROL-w	Kill from the cursor to the mark.
<i>kill</i>	The <i>kill</i> character (CONTROL-u in the A/UX standard distribution). Kill the entire current line. If two <i>kill</i> characters are entered in succession, all <i>kill</i> characters from then on cause a line feed (useful when using paper terminals).
CONTROL-y	Restore last item removed from line. (Yank item back to the line.)

5.2.5 Other *emacs* editing commands

These miscellaneous commands are also available:

CONTROL-l	Line feed and print the current line.
CONTROL-@	(null character) Set mark.
<i>M-(space)</i>	(meta-space) Set mark.
CONTROL-j	(newline) Execute the current line.

- CONTROL-m (return) Execute the current line.
- M-p* Push the region from the cursor to the mark on the stack.
- M-digits* (escape) Define numeric parameter; the digits are taken as a parameter to the next command. The commands that accept a parameter are *.*, CONTROL-f, CONTROL-b, *erase*, CONTROL-d, CONTROL-k, CONTROL-r, CONTROL-p, CONTROL-n, *M-.*, *M-__*, *M-b*, *M-c*, *M-d*, *M-f*, *M-h*, and *M-CONTROL-h*.
- CONTROL-u Multiply parameter of next command by 4.

5.3 Using **fc** or **x**

Another way to access and edit the commands listed in your `.sh_history` file is to use the `fc` command. The `fc` command uses the value of the `FCEDIT` environment variable as its editor; this is set to `/bin/ed` by default. See “Commonly Used Environment Variables” for more information.

5.3.1 Editing and reexecuting previous commands

In the command

```
fc -e - string=new-string
```

the option “`-e -`” means that you wish to execute a command indicated either by *string* or by its number. If it is indicated by *string*, the most recent command with those characters will be selected. If *string=new-string* is included, *new-string* replaces *string* before execution. If the command is specified by number and it does not include *string*, the shell displays the message

```
bad substitution
```

and the `fc` command fails. For example, the command

```
fc -e - vi
```

reexecutes your most recent `vi` command. If you want to substitute another filename to your most recent `vi` command, you can use a command such as

```
fc -e - chap1=chap2 vi
```

An abbreviated form of

```
fc -e -...
```

is the command

```
r old=new command
```

This command works exactly like the `fc` command and is provided simply because it is easier to type. For example, to edit and reexecute the `vi` command discussed above, you type

```
r chap1=chap2 vi
```

The command

```
r command >file
```

reexecutes *command* with the output directed into *file*.

To edit command(s) with `fc`, use the form

```
fc first last
```

```
fc string
```

The command(s) specified by the range *first* to *last* (or the command that begins with *string*) are copied into a temporary file, and the editor named by the `FCEDIT` variable is invoked.

Once you are in the editor, you can use any of its commands. When you exit, your edited command or commands are read by the Korn shell and executed. As each command is executed, it is printed at the terminal.

For example, to edit and reexecute the list of commands

```
15 cp chap1 chap1.bck
16 lp chap1
17 mv chap1 /printed
```

you give the command

```
fc 15 17
```

After this command, you see these commands displayed and can edit them as you desired with any editor command (for example, replacing the `1` in `chap1` with the number `2`). When you exit the editor, the new commands are executed and entered in the history file.

Likewise, to edit and reexecute the last `diff` command you gave, you can use the command

```
fc diff
```

Finally, you can also use the `fc` command without using an editor. This can be useful when you want to reexecute a command without changing it, or when you wish to make a simple change and do not want to spend the time necessary to use an editor.

5.3.2 Listing previous commands

With the `-l` option, `fc` accepts command numbers or strings as arguments. With command numbers

```
fc -l first last
```

`fc` prints a list of commands, where *first* is the oldest command you wish to review and *last* is the most recent. For example,

```
fc -l 10 12
```

first and *last* may also be negative numbers:

```
fc -l -10
```

A negative number is interpreted as the *n*th previous command. If *first* is given but not *last*, then commands from *first* through the current command are listed. If no numbers are specified, the 16 most recent commands are listed.

If you ask for commands that are not available, either because the command is too old (remember that only the number of commands given in `HISTSIZE` are saved) or because you have not given that many commands, the shell will display the message

```
Bad number.
```

“`fc -l`” can be combined with two other options:

- r List specified commands from most recent to oldest.
- n List specified commands without command numbers.

For example, the command

```
fc -lr 10 12
```


prints command numbers 10, 11, and 12 from your history file in reverse order. The output might look like this:

```
12 vi chap2.ksh
11 ls chap*
10 rm chap2.bck
```

With *string* as an argument to `fc -l`:

```
fc -l string
```

you can search for and print a list of commands beginning with a command containing *string*. For example, to obtain a list from your most recent `rm` command to your current command, you could type

```
fc -l rm
```

6. Using shell metacharacters

Shell **metacharacters** are characters that perform special functions in the shell. This section discusses how to use these metacharacters. The following are the Korn shell metacharacters:

- ~ A tilde is used as the first part of a directory name. It is replaced with either your home directory (if it is used alone or followed by a pathname below your home directory such as `~/project/phase1`) or the home directory of another user (if it is followed by the login name of that user, such as `~lori`). See “Specifying Home Directories” for details.
- & An ampersand at the end of a command line causes the shell to run the command(s) in the background and prints the process ID(s).
- ? A question mark used as part of a file or directory name causes the shell to match any single character (except a leading period).
- * An asterisk used as part of a file or directory name causes the shell to match zero or more characters (except a leading period).
- [] Brackets around a sequence of characters (except the period) cause the shell to match each character one at a time.
- A hyphen used within brackets to designate a range of characters (for example, `[A-Z]`) causes the shell to match each character in

the range.

- < A less-than sign following a command and preceding a filename causes the shell to take the command's input from that file.
- > A greater-than sign following a command and preceding a filename causes the shell to redirect the command's standard output into the file. See "Input and Output" for a description of how this metacharacter is used to redirect error output.
- >> Two greater-than signs following a command and preceding a filename cause the shell to append the command's output to the end of an existing file.
- | A vertical bar between two commands on a command line causes the shell to redirect the output of the first command to the input of the second command. This can occur multiple times on a command line, forming a pipeline.
- | & A vertical bar and ampersand at the end of a command cause the shell to connect this background command to the parent shell (and the terminal, if this shell's output and input is connected to the terminal). Output and input can be read and written to the background process. See "Connecting a Command to Standard Input and Output."
- ; A semicolon between two commands on a command line causes the shell to execute the commands sequentially in the order in which they appear.
- () Parentheses around a pipeline or sequence of pipelines cause the whole series to be treated as a simple command (which may in turn be a component of a pipeline), and a subshell to be spawned for the commands' execution.
- { } Braces around a series of commands group the output of the commands.
- \ A backslash prevents the shell from interpreting the metacharacter that follows it.
- ' ' Single quotes around a command, a command name and argument, or an argument prevent the shell from interpreting the enclosed metacharacters.

- " " Double quotes around a command, a command name and argument, or an argument prevent the shell from interpreting the enclosed metacharacters, but only as follows: file, wildcard, and command substitution will take place, but filename expansion and interpretation of blanks will not.
- \ ` Back quotes around a command cause the characters in that command to be replaced with the output from that command.

6.1 Shortcuts in working with directories

6.1.1 Specifying home directories

You can use the tilde (~) as the initial character in a filename or pathname to avoid typing the absolute or relative pathnames of home (login) directories. An initial tilde in a pathname, for example,

```
~/chapter2
```

indicates a file below your own home directory. When the command is executed, the tilde is replaced by the value of your environmental variable HOME. A tilde followed by the login name of another user, for example,

```
~virginia
```

indicates the login name of that user, and will be replaced by the absolute pathname of that user's home directory.

You can use this notation when giving a pathname as an argument to any command; for example,

```
cp ~virginia/memo1 ~/memos/virginia.memo
```

6.1.2 Current and previous directories

The tilde can also be used to represent your current and previous working directories. A tilde followed by a plus sign (+) represents the current working directory (the value of the parameter PWD); tilde followed by a minus sign (-) is replaced by the most recent working directory (the value of the parameter OLDPWD).

For example, use the `cd` command to return to your most recent working directory with the command

```
cd ~-
```

You can toggle between two directories by repeating this command several times.

6.1.3 Substituting directory names

The Korn shell also allows substitution on directory names as arguments to the `cd` command

```
cd old new
```

where the *new* directory name replaces *old* in the full pathname of the current working directory (the parameter `PWD`). For example, suppose you had the directories

```
/users/doc/anne/manuals/drafts  
/users/doc/anne/manuals/review1  
/users/doc/steve/manuals/review1
```

After the command

```
cd /users/doc/anne/manuals/drafts
```

you could go to `/users/doc/anne/manuals/review1` with the command

```
cd drafts review1
```

From there, you could then go on to

```
/users/doc/steve/manuals/review1
```

with the command

```
cd anne steve
```

Each time you change to a directory using “`cd` substitution,” the full pathname of the new directory is displayed.

6.2 Specifying filenames with metacharacters

Using the filename expansion metacharacters (also called “wildcards”) will spare you the job of typing long lists of filenames in commands, looking to see exactly how a filename is spelled, or specifying several filenames that differ only slightly.

These metacharacters are interpreted and take effect when the shell evaluates commands. At this point, the word incorporating the metacharacter(s) is replaced by an alphabetic list of filenames, if any

are found that match the pattern given. Filename expansion metacharacters can be used in any type of command; however, in the case of filenames given for input and output redirection, filename expansion may cause unexpected results if the metacharacter usage expands into more than a single filename. To turn off the special meaning of metacharacters and use them as ordinary letters, they must be quoted. See “Quoting.”

The following are filename expansion metacharacters in the Korn shell:

- ? A question mark matches any single character in a filename. For example, if you have files named

```
a bb ccc dddd
```

the command

```
print ???
```

matches a sequence of any three characters and returns

```
ccc
```

- * An asterisk matches any sequence of characters, including the empty sequence, in a filename. (It will not, however, match the leading period in such files as `.profile`.) To list the sequence of files named

```
chap chap1 chap2 chap3 chap3A chap12
```

you can use the notation

```
ls chap*
```

The files are listed as

```
chap chap1 chap12 chap2 chap3 chap3A
```

Note that in the first file listed, `chap`, the asterisk matched the null sequence composed of no characters.

- [] Brackets enclosing a set of characters match any *single* character, one at a time, from the set of enclosed characters. Thus,

```
ls chap.[12]
```

matches the filenames

```
chap.1 chap.2
```

Note that this does not match `chap.12`. To match filenames `chap.10`, `chap.11`, and `chap.12`, use the notation

```
chap.1[012]
```

You can also place a hyphen (-) between two characters in brackets to denote a range. For example,

```
ls chap.[1-5]
```

is the equivalent of

```
chap.[12345]
```

A range of characters can also be indicated in brackets. The notation `[a-z]` matches any lowercase character, `[A-Z]` matches any uppercase character, and `[a-zA-Z]` matches any character, regardless of case.

To match anything *except* a certain character or range of characters, use the exclamation point inside the brackets. When the first character following the left bracket ([]) is an exclamation character (!), any character *not* enclosed in the brackets is matched. For example,

```
[!b]
```

matches any filename composed of one letter, except a file named `b`.

None of these metacharacters will match the initial period at the beginning of special files such as `.profile`. These must be matched explicitly. Periods that do not begin a filename can be matched by metacharacters.

If you use these metacharacters and the shell fails to match an existing filename, it displays a message such as

```
ksh: *: not found.
```

6.3 Input and output redirection

An executing command may expect to accept input and create output, possibly including error output (error messages). In the A/UX system, there are default locations set for input and output:

- Standard input is taken from the terminal keyboard.
- Standard output is printed on the terminal screen.
- Standard error output is printed on the terminal screen.

These defaults can be changed using the following metacharacters (also called **redirection symbols**). The redirection metacharacters are a way of using file descriptors, described in detail in “Redirection With File Descriptors.”

- < A less-than sign followed by a filename “redirects standard input” (takes command input from a file or device other than the keyboard). For example,

```
mail ellen < note
```

uses a file named `note` instead of a message typed from the keyboard as the input to `mail`.

- > A greater-than sign followed by a filename “redirects standard output” (prints command output in a file or to a device other than the terminal screen). If a file by that name already exists, its previous contents are overwritten; otherwise a new file is created. For example,

```
sort file1 > file2
```

uses a file for the output of the `sort` command. When `sort` completes, `file2` contains the sorted contents of `file1`.

See “Input and Output” for information on redirecting standard error output using file descriptors.

- >> Two greater-than signs followed by a filename append the output of a command to a file. If no file by that name exists, one is created. For example,

```
who >> log
```

appends the output of the `who` command to the end of the existing file `log`.

6.4 Combining commands: pipelines

You can send the output of one command as input to another command by using the vertical bar or “pipe” (`|`). When two or more commands are joined by a pipe, the command line may be considered a **pipeline**.

For example, to see which files in a directory contain the sequence `old` in their names, you can use a pipeline as follows:

```
ls | grep old
```

The pipe character (`|`) tells the shell that output from the first command (the list of files produced by the `ls` command) should be used as input to the `grep` command. The output of the pipeline (filenames in the current directory containing the string `old`) prints on standard output (unless you redirect it to a file).

Pipelines may consist of more than two commands; for example,

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

prints the number of files in the current directory whose names contain the string `old`.

Pipelines may also be executed in the background. For example, to avoid the time-consuming process of waiting for a very large file to be sorted and printed, you could give the following pipeline:

```
sort mail.list | lp &
```

This pipeline would sort the contents of a file named `mail.list` and send the sorted information to the `lp` program to be placed on the printer queue. The shell would respond with the process ID of the last command in the pipeline.

The `tee` command is a “pipe fitting”; it can be put anywhere in a pipeline to copy the information passing through the pipeline to a file. See `tee(1)` in *A/UX Command Reference* for more information.

A **filter** is a program or a pipeline that transforms its input in some way, writing the result to the standard output. For example, the `grep` command finds those lines that contain some specified string and prints them as output.


```
grep 'correction' draft1
```

prints only the lines in `draft1` that contain the string `correction`.

Filters are often used in pipelines to transform the output of some other command. For example,

```
who | grep jon
```

prints

```
jon          ttyt8          Jul 21 12:25
```

if a user whose login name is `jon` is currently logged into the system on `ttyt8`.

6.5 Connecting a command to standard input and output

In the Korn shell, the input and output of a command or pipeline running in the background can be connected to standard input and output by ending the command line with `|&`. This establishes a two-way pipe with the shell.

Output created by the background process can then be read using the `read -p` command as follows:

```
read -p variable
```

The input line from the pipe will be read into *variable* and then used as desired.

Input for the pipe can be inserted with the `print -p` command:

```
print -p arguments
```

The arguments are written onto the pipe for use by the background process.

Only one background process connected to the shell with `|&` can be running at a time. For example,

```

cat |&
    [1] 6420
print -p "hello"
print -p "goodbye"
read -p var
echo $var
    hello
    read -p var
echo $var
    goodbye

```

where the indented lines show output printed on the terminal.

6.6 Command grouping

You can use the following metacharacters to group commands together:

- ;** Group several commands on one command line by separating one command from another with a semicolon (;). The commands will be executed sequentially in the order in which they appear. For example, the command line

```
cd test; ls
```

will change to the `test` directory and then list its contents.
- &** Group background commands on a single line by separating them with ampersands (&) and then ending the line with another ampersand. The background commands will exit independently while the shell continues to accept new commands in the foreground.
- { }** Use braces to group commands for functions and control-flow constructs (see “Defining Functions” and “Control-Flow Constructs”). You can also use braces to group the output from several sequential commands, which is then used as the input to a following command in a pipeline. Braces used in the latter way are recognized only when they are the first word of a command or are preceded by a semicolon or newline, and when the first brace is followed by a space. For example, to put the date and the list of users into one file (`log`), you can give the command

```
{ date; who;} | cat > log
```

Note the space following the first brace and the semicolon following the last command in braces; these are required. If you type a newline before closing with another brace, you will see the secondary prompt until you give the closing brace. Note that commands enclosed in braces are executed by the current shell (that is, a new instance of the shell is not invoked to execute them).

- () Enclose a group of commands in parentheses to execute them as a separate process in a subshell (a new instance of the shell). For example,

```
(cd test; rm junk)
```

first invokes a new instance of the shell. This shell changes the directory to `test` and then removes the file `junk`. After this, control is returned to the parent shell, where the current directory is not changed. Thus, when execution of the commands is over, you are still in your original directory.

The commands

```
cd test; rm junk
```

(without the parentheses) are executed in the current shell and have the same effect but leave you in the directory `test`.

6.7 Conditional execution

You can use the following symbols to indicate that your command should be executed only if some condition is met:

&& The command form

```
command1 && command2
```

means “If *command1* executes successfully (returns a zero exit status), then execute *command2*.”

|| The command form

```
command1 || command2
```

does the reverse. This form means “If *command1* does not

execute successfully (returns a nonzero exit status), then execute *command2*.”

For exit status, see “Exit Status: The Value of the Command.”
Conditional execution is also available in joining pipelines. For other ways of obtaining conditional execution, see “Control-Flow Constructs.”

6.8 Quoting

If you need to use the literal meaning of one of the shell metacharacters or control the type of substitution allowed in a command, use one of the following **quoting mechanisms**:

- \ A backslash preceding a metacharacter prevents the shell from interpreting the metacharacter. For example, to use the `print` command to display a question mark, you must precede the question mark with a single backslash (`\`). Thus,

```
print \?
```

prints

```
?
```

Without the backslash, the `print` command would generate a list of all one-character filenames in the current directory. If there are none, the command returns

```
?
```

- ' ' Single quotes prevent the shell from interpreting any metacharacters in the enclosed string. The command

```
print '$EDITOR'
```

prints

```
$EDITOR
```

- " " Within double quotes, parameter substitution and command substitution occur, but filename expansion and the interpretation of blanks do not. For example, the command

```
print "$EDITOR"
```

prints

```
/bin/ed
```

Here parameter substitution filled in the value of the environmental variable EDITOR.

Double quotes can also be used to give a multiword argument to commands; for example,

```
print "type a character"
```

For more information on parameter substitution, see “Positional Parameters and Shell Variables.” You can also suppress filename expansion universally by setting the shell option `-f`; see “Shell Execution Options.”

- ‘ ‘ A command name enclosed in back quotes is replaced by the output from that command. This is called **command substitution**. For example, if the current directory is `/users/marylyn/bin`, the command

```
i=`pwd`
```

is equivalent to

```
i=/users/marylyn/bin
```

If a back quote occurs within the command to be executed, you must escape it with a backslash (`\ ``); otherwise the usual quoting conventions apply within the command.

Command substitution takes place before the filenames are expanded. If the output of substituted command is likely to be more than one word, the command must be enclosed in double quotes as well as back quotes; for example,

```
a="`head -1 /dev/tty`"
```

where the command `head -1` (read the first line of input) might yield more than one word.

7. Working with more than one shell

When you wish to use another A/UX shell, you can use one of the following commands:

`sh` This spawns an instance of the Bourne shell.

`ksh` This spawns another instance of the Korn shell.

`csh` This spawns an instance of the C shell.

You can type these at your shell prompt; for example,

```
ksh
```

In this case, your new shell will run as a **subshell** or “child” of your current one. You can use the `exit` command or the `eof` sequence to return to your original login shell whenever you wish. (If you accidentally give the `exit` command or send an `eof` in your login shell, you will be logged out of the system altogether.)

7.1 Changing to a new shell

You can also obtain a new shell using the `exec` command; for example,

```
exec csh
```

If you use the `exec` command, the C shell program `csh` replaces your current shell. You cannot return to your original shell; it has disappeared.

Generating new instances of a shell affects the environment settings for each shell. See “The Environment and New Shell Instances” for more information.

7.2 Changing your default shell

To change your default shell from the Korn shell to the Bourne or C shell, use the `chsh` command. For example,

```
chsh login.name /bin/csh
```

(where *login.name* is your login name on this system) changes your default login shell to the C shell. See `chsh(1)` in *A/UX Command Reference* for more information.

8. The environment

The **environment** is a list of variables, aliases, and functions that is available to all programs (including subshells) invoked from the shell. A shell inherits the environment that was active when it started, and passes that environment (including any modifications) to all programs

it invokes.

If you assign values to variables using the `typeset` command at the shell prompt (or within a shell script), these remain local to the shell in which you assigned them. If you use the `typeset -x` command (or set the `-a` shell option; see “Shell Execution Options”), these changes will be passed on to any subshells you invoke and to executing commands.

Note: Modifying the environment in a subshell (for example, in a shell script) never changes the parent shells or their environments. Because these changes are made to a *copy* of the parent shell’s environment, the parent shell’s environment is never affected by changes in a subshell, even if you use the `export` command. When a subshell terminates, its environment no longer exists.

In general, the most essential variables are assigned default values during login or by the shell every time you invoke it. The Korn shell also defines a number of default aliases (see “Aliases for Commonly Used Commands”). Convenient but inessential variables are simply left unassigned. Thus a default environment is created for you. You can modify the default environment by defining new environment variables and aliases.

8.1 Listing existing values

Table 3-1 shows commands you can use to list existing values in the environment.

Table 3-1. Listing functions, aliases, and variables

Command	Output
<code>set</code>	lists everything defined
<code>env</code>	lists exported variables
<code>export</code>	lists exported and read-only variables
<code>typeset</code>	lists all variables
<code>typeset option</code>	lists variables of <i>option</i> type
<code>typeset -f</code>	lists functions

```
typeset -x    lists exported variables and functions
alias        lists aliases
alias -x     lists exported aliases
```

8.2 Assigning values to environment variables

Setting up your own customized environment is not necessary, but it can make your work easier and more efficient. To customize your working environment, you may change the default values assigned to some of your environment variables and add others that have not been included.

Unless you have set the `-a` shell execution option (which tells the shell to export all variables automatically; see “Shell Execution Options”), you assign a value to an environment variable using the command

```
typeset -x name=value
```

This command sets the variable *name* to *value* and automatically inserts the variable and its value in the environment. Thus, for example, to assign and export the variable `HISTFILE` you could give the command

```
typeset -x HISTFILE=/users/daphne/hist
```

In addition to the `typeset -x` command, the Korn shell also recognizes the Bourne shell syntax:

```
name=value
export name
```

This is the form that should be used in `.profile` if you are ever going to log into the Bourne shell.

8.3 Removing environment variables

The command

```
unset name
```

removes the specified variable.

8.4 Commonly used environment variables

The following variables are typically inserted into the environment. By convention, environment variable names are uppercase. Some of these variables are assigned default values at login or by the shell at

invocation. All of them can be reset by the user.

The variables used only by the Korn shell are as follows:

COLS	This variable defines the width of the edit window for the inline editing. The default is 80 columns.
EDITOR	This variable and the VISUAL variable specify the editor for inline editing of commands. The default is ed. This is the same as setting the option -o ed with the set command.
ENV	This variable specifies the name of the Korn shell environment file. If this variable is set to a filename and exported in the /etc/profile system file (it is initially set to \$HOME/.kshrc and exported on A/UX systems), then all subsequent instances of the Korn shell read the specified filename when the shell starts up. The ENV file is typically used to set up inline command editing and command reuse, and for alias and function definitions. Command and parameter substitution are performed in referencing this variable.
FCEDIT	This variable specifies the editor for the command reentry with the fc command. The default editor is ed.
HISTFILE	This variable gives the pathname of the file to be used to store command history for command reentry. The default filename is \$HOME/.sh_history, that is, a file named .sh_history in your home directory.
HISTSIZE	This variable specifies the number of previously entered commands that will be saved for command reentry.
PS3	This variable gives the prompt to be used by the select command after a menu is given. The default is "#?".
VISUAL	This variable specifies the visual editor to use in line-editing mode. Initially, this variable is unset.

The variables used by all shells follow:

CDPATH	The value of this variable should contain a list of pathnames (separated by colons) that you use frequently. The shell uses this variable when you give an argument to the <code>cd</code> command that is not a relative or absolute pathname. This variable is usually set in the <code>.profile</code> file; otherwise its default value is the current directory.
EXINIT	This variable indicates various options for your editing environment when you are using the <code>ex</code> or <code>vi</code> text editing program (see “Using <code>ex</code> ” and “Using <code>vi</code> ” in <i>A/UX Text Editing Tools</i>).
HOME	This variable specifies your home directory. The login procedure sets the value of this variable to the pathname of your login directory.
IFS	The shell uses this variable to interpret blanks. The default values of this variable are space, tab, and newline, specifying the characters used to separate the parts of commands. You can reset this to include any data delimiters.
MAIL	The shell uses this variable as the pathname of the file where your mail is delivered. This variable is typically set in the file <code>.profile</code> in the user’s login directory.
MAILCHECK	This variable specifies how often (in seconds) the shell will check for the arrival of mail in the file specified in <code>MAIL</code> . The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.
PATH	The value of this variable should be a series of pathnames separated by colons (:). The shell uses the value of <code>PATH</code> executable programs whenever you give a command. If the directory containing the command is not specified, the shell will display the message <pre>Command not found.</pre> <p><code>PATH</code> is usually set in the <code>.profile</code> file. For efficiency, the list of directories in the <code>PATH</code> variable should be in order from the directories containing</p>

	commands most often used to those least often used. The default value for <code>PATH</code> is the current directory, <code>/bin</code> , and <code>/usr/bin</code> .
<code>PS1</code>	This variable specifies the primary prompt string (the prompt you see when the shell is waiting for you to give a command). The default setting is the dollar sign (<code>\$</code>).
<code>PS2</code>	This variable specifies the secondary prompt string (the prompt you see when the shell is waiting for more information for a command you have already started). The default setting is the greater-than sign (<code>></code>).
<code>SHELL</code>	This variable specifies your login shell. It is set at login to the value found in the <code>/etc/passwd</code> file. If no shell is specified in <code>/etc/passwd</code> , the value of <code>SHELL</code> is <code>/bin/sh</code> . For instructions on how to change your login shell, see <code>chsh(1)</code> in <i>A/UX Command Reference</i> .
<code>TERM</code>	This variable specifies the type of terminal you are using. The default value is <code>mac2</code> . You can find out what your current terminal type is with the command <pre>print \$TERM</pre>
<code>TZ</code>	This variable indicates your time zone. It is set at login.

8.5 The environment and new shell instances

If the `ENV` variable is set and exported, the Korn shell reads the contents of the file (initially set to `$HOME/.kshrc`) every time it starts up. Thus, the values you have defined there are available to every new instance of the Korn shell. Any values you have assigned using the `typeset -x` command are in the environment and will be available to new shell instances.

If you have assigned values to variables using the `set` command at the shell prompt (or within a shell script), these remain local to the shell in which you assigned them. Because these changes are made to a *copy* of the parent shell's environment, the parent shell's environment is never affected by changes in a subshell, even if you use the `typeset -x` command in the subshell. Note, however, that changes made using `typeset -x` in a subshell will be passed on to new instances invoked

from the subshell. When a subshell terminates, its environment no longer exists.

Note that the `.profile` file is read only once, at login. Thus, if you have changed the value of an environment variable, the subshell will inherit the new value, not the value set routinely in `.profile`. You can force a new instance of the shell to read `.profile` by using the “dot” command (`.`); see “Executing Shell Scripts.”

8.6 Special environments

Normally, the environment for a command is the complete environment of the shell where the command was given. You can change the environment used by a command in three ways:

- Augment the environment by inserting additional variables and new values into the environment. This is done by preceding the command with one or more assignments to variables on the command line. For example,

```
a=b command
```

Note that because parameter substitution occurs before the environment is changed, you *cannot* assign environment variables whose values are then immediately referenced on the command line. For example, the sequence of commands

```
x=5
x=3 print $x
prints
5
not
3
```

because the value of `x` is inserted into the command line before the environment is changed.

- Set the `-k` shell option using the command

```
set -k
```

When `set`, this shell option inserts variables and values given on the command line into the environment for a particular

command. For example, if the `-k` option is *not* set, the command

```
print a=b c
```

prints

```
a=b c
```

After `-k` has been set, `a=b` is interpreted as a variable assignment instead of an argument, and the same command prints

```
c
```

Note that because values are substituted for variables before the environment is changed, this is subject to the same limitation documented above.

- Use the `A/UX` command

```
env [-][name=value ...][command][args]
```

to set the environment for the command. With this command, you can not only add things to the environment inherited by a command, but also exclude the current environment. To add variables and their values to the current environment, give the variables and values before the command name. For example, to run a subshell with a changed `PATH` environment variable, you could give the command

```
env PATH=directory-list sh
```

For the duration of the new shell (and its subshells), the `PATH` variable would be set to the directories in the list.

To set up a completely new environment, first give the option `-`, which excludes the current environment, and then assign the variables and values you want. These (and only these) will be available in the environment for the new command.

8.7 The default environment on your system

Whenever you log in, the following procedures occur:

- The `login` program sets the variables `HOME` and `SHELL` from the information in the system file `/etc/passwd`.

- The `login` program then checks the file `/etc/profile` to find out the default environment to set up for all users. This file may contain default settings for `PATH`, `TZ`, and `TERM`.
- The **login shell** (the shell that is automatically invoked when you log in) assigns default values to `PS1` (the primary prompt), `PS2` (the secondary prompt), `PS3` (the prompt for the `select` command), `MAILCHECK`, and `IFS` (Input Field Separator, which can be blank characters and/or tabs).

When you invoke new instances of the shell (for example, using the `ksh` command), the new shell checks the environment for any new values you may have placed there for these variables. If it doesn't find any values in the environment, it assigns the default values.

Then the new shell reads your `.profile` file. If you have assigned new values there, it uses your values instead of the defaults.

If the `ENV` variable is assigned a filename and exported, whether in the `/etc/profile` system file or in the `.profile` file in your home directory, the new shell reads the contents of that file and sets the values you have assigned there.

- The Korn shell reads the `.profile` file when you log in; if appropriate, it shares the variable assignments with the Bourne shell.
- If the `ENV` variable is assigned a filename and exported, whether in the `/etc/profile` system file or in the `.profile` file in your home directory, the Korn shell reads the contents of that file every time it starts up. This is initially set to `$HOME/.kshrc` on most systems; in this case, use the `.kshrc` file in your home directory to set the environment variables unique to the Korn shell and to define aliases you wish to be available across invocations of the shell.

9. The `.profile` file

The `.profile` file is simply a text file (created with a text editor). It contains a series of commands typed exactly as you would type them at the shell prompt. Every time you log in, the shell looks in your home directory for a file named `.profile` and executes all the commands found there before issuing the shell prompt and taking commands. If

no `.profile` file exists, your environment will simply be the default environment created by the shell at login.

9.1 A sample `.profile` file

The following is a sample `.profile` file:

```
typeset -x PATH=:/bin:/usr/bin:$HOME
typeset -x CDPATH=:/users/elaine/versions
typeset -x MAILCHECK=0
typeset -x EXINIT="set wm=10"
date
ls
```

Note: You may also use the Bourne shell style `.profile` using the `set` and `export` commands. See “A Sample `.profile` File” in Chapter 2, “Bourne Shell Reference.”

The variables and commands in this file are discussed in the sections that follow. In theory, *any* A/UX command or shell script may be invoked in the `.profile`; typically, however, you should include commands that customize your login shell or perform login initialization routines (such as listing the contents of the current directory, or reading your mail). Commands you want to affect *all* subshells of the login shell should be put into the file assigned to the ENV variable (usually the `.kshrc` file). See “The `.kshrc` File.”

9.1.1 Locating commands

The `PATH` environment variable lists the directories (separated by colons) where the shell will look for the executable files that are A/UX (or user-defined) commands. Each time you give a command, the shell searches the directories listed in the order specified. Most A/UX commands are located in the `/bin` or `/usr/bin` directory. When you assign a value to `PATH`, be sure to include these directories.

If the shell cannot find the file in one of the directories specified, the command cannot be executed and you will see the message

```
Command not found.
```

The directories listed in the `PATH` variable are specified by their absolute pathnames, separated by colons. If the list of directories begins with a colon, the path search begins in the current directory. At login, the `PATH` variable might be set as follows:

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

This assignment sets the `PATH` variable to the current directory and the system directories `/bin`, `/usr/bin`, and `/usr/ucb`.

To reset the `PATH` variable in `.profile`, insert lines such as

```
typeset -x PATH=:/bin:/usr/bin:/usr/ucb:$HOME
```

The `typeset -x` command is discussed in “Customizing Your Environment.”

If you include the pathnames of personal directories that contain shell programs you have written, these will be accessible to the shell no matter what your current directory is. If you wish to execute a command or shell program that is not in one of the directories in your `PATH` variable, simply give the absolute pathname of the directory where the command or shell program is to be found.

For information on referencing variables using the `$` syntax (as in `$HOME` above), see “Parameter and Variable Substitution.” For more information about pathnames, see the glossary in *A/UX System Overview*.

9.1.2 Shortcuts in changing directories

If `CDPATH` is set, you can use the `cd` command with a simple directory name that is neither an absolute nor a relative pathname. The shell then searches for that directory in all the directories listed in `CDPATH`. The directories are searched in the order specified. If `CDPATH` is not set, only the current directory is searched.

If the directory you specify is not found in any of the directories given in `CDPATH`, you will see the message

```
Bad directory.
```


After `CDPATH` is set, you can still, of course, give the relative or absolute pathname of any directory you wish. When you give an absolute or relative pathname in the `cd` command, `CDPATH` is not used.

9.1.3 Receiving mail

The `MAILCHECK` environment variable specifies how often (in seconds) the shell should check for new mail. When you log in, the shell sets `MAILCHECK` to 600 seconds (10 minutes). You can change this to whatever period you wish using the command

```
typeset -x MAILCHECK=0
```

This command assigns and exports the value of the `MAILCHECK` as 0. When `MAILCHECK` is 0, the shell checks for new mail before each prompt.

9.1.4 Your editing environment

The `EXINIT` environment variable tells the shell how to initialize the `vi` or `ex` editing program. This variable is set to a series of editor commands that should be run every time the editor is called before any commands are read from the terminal. In the sample `.profile` above, for example, the command

```
typeset -x EXINIT="set wm=10"
```

assigns and exports the value of `EXINIT` as the command

```
set wm=10
```

which sets the word-wrap margin so that the editor will automatically break lines ten spaces before the right margin. The command is enclosed in double quotes because the entire string must be treated as one “word” and not divided up.

For details on `EXINIT`, see *A/UX Text Editing Tools*. For the use of double quotes, see “Quoting.”

9.2 Customizing your login procedure

You can also use your `.profile` file to customize your login procedure. In the sample `.profile` above, the commands

```
date
ls
```

direct the shell to display the date and time and then list all the files in the current directory before displaying the shell prompt. These will be executed at login.

You can include any commands you wish in `.profile`, including your own functions and shell scripts.

10. The `.kshrc` file

A/UX systems use the `/etc/profile` system file to define the `ENV` variable to a filename and export this variable. On A/UX systems this is initially set to `$HOME/.kshrc`, but this may be changed to another filename by modifying the value of the `ENV` variable. See “Changing the Name of the `ENV` File.”

If this variable is set to any filename and exported, that file will be read whenever the Korn shell starts up. Thus, any definitions you include in the file named as the `ENV` file (initially `$HOME/.kshrc`) will be available to every instance of the Korn shell. You can create a `.kshrc` file in your home directory and use it to define variables, aliases, and functions that are applicable only to the Korn shell.

Note: If the `ENV` variable is not defined as `$HOME/.kshrc` and exported, the Korn shell will not read your `.kshrc` file.

For information on aliases, see “Aliases for Commonly Used Commands.” For functions, see “Defining Functions.”

10.1 A sample `.kshrc` file

The following is a sample `.kshrc` file:

```
typeset -x HISTFILE=/users/neal/my.history
typeset -x HISTSIZE=15
```

These commands are described below.

10.1.1 Changing history variables

The sample `.kshrc` file resets the following variables:

HISTFILE This variable specifies where the text of past commands should be stored. The default file is `.sh_history` in your home directory. The command

```
typeset -x HISTFILE=/users/neal/my.history
```

assigns and exports the value of the `HISTFILE` as the file named `my.history` in the directory `/users/neal`.

HISTSIZE This variable specifies how many past commands should be saved. The command

```
typeset -x HISTSIZE=15
```

assigns and exports the value of the `HISTSIZE` as 15. After this command, only 15 past commands would be saved.

10.2 Changing the `ENV` filename

The A/UX system defines the `ENV` variable to `$HOME/.kshrc` in the system file `/etc/profile`. This assigns this variable a value when you log in.

To change the name of this file, you can reset `ENV` in your `.profile` file; for example,

```
typeset -x ENV=filename
```

or

```
ENV=filename  
export filename
```

11. Aliases for commonly used commands

The Korn shell `alias` command renames existing commands or creates a name for a long command line. Aliases may be defined at the shell prompt or in the `.kshrc` file.

Note: The Korn shell also provides a facility for defining functions. This is similar to aliasing and may be preferable for some of your tasks. See “Defining Functions.”

The Korn shell keeps a list of aliases. Each time you give a command, the first word of the command is compared with the list. If it is an alias name, then it is replaced with the definition of that alias. You can use an alias to redefine any shell or A/UX command; however, you cannot redefine keywords such as `if` or `done`.

11.1 Defining an alias

You define an alias with the command

```
alias name=definition
```

where *name* may begin with any printable character, but the rest of the characters must be letters, digits, or underscores (generally it is a good idea to avoid using `/`, `;`, `*`, `?` and so on); `=` cannot be surrounded by blank spaces; and *definition* may contain any valid commands, including shell scripts and metacharacters. Note that *definition* cannot include another alias. If *definition* includes spaces, the whole command must be inclosed in quotes.

For example, the alias

```
alias ls='ls -C'
```

causes the `ls` command to produce output as if you had typed

```
ls -C
```

which displays its output in columns. The alias definition is quoted because it contains a blank. In the example above, every time you type `ls`, you will get `ls -C`, and this may not be desirable. It is recommended that you invent a new command name, as in

```
alias lc = 'ls -C'
```

This allows you to use both `ls` (in any form desired) *and* `lc`.

Alias definitions can also include all shell metacharacters, variables, positional parameters, command substitution, and so forth.

For example,

```
alias prtsort='sort *.list'
```

creates a command `prtsort`. When you type

```
prtsort
```

the command line

```
sort *.list
```

executes, sorting files in the current directory that end in the characters “.list”.

When you create aliases at the shell prompt, they are not exported to the environment unless you use the `-x` option:

```
alias -x lc='ls -C'
```

Exported aliases remain in effect for subshells but must be reinitialized for separate invocations of the shell. To make aliases available to every invocation of the Korn shell or any script run with a separate shell, put their definitions in the `.kshrc` file, which is read every time a Korn shell is started up.

Note: Aliasing is performed when scripts are read, not while they are executing. Therefore, for an alias to take effect, the `alias` command has to be executed before the command that references the alias is read.

11.2 Listing and removing aliases

The `alias` command with no arguments lists all aliases that have been defined in your environment. To list the text of exported aliases, use the `alias -x` command.

Aliases can be removed with the command

```
unalias name(s)
```

11.3 Tracking with aliases

Aliases invoked with the `-t` option are used to reduce the amount of time the shell spends searching the directories specified by the `PATH` variable for a particular command. This is called **tracking**: when you use a “tracked” command, it is treated like an alias that corresponds to the full pathnames of that particular command. For example, if you give the command

```
alias -t sort
```

the shell interprets `sort` as an alias for the full pathname of the `sort` command (`/bin/sort`). After you have used the above command, `sort` is defined as the following alias:

```
alias sort=/bin/sort
```

This allows the shell to substitute the full pathname and bypass the directory search specified in your `PATH` variable.

Note that the same effect can be produced for all A/UX commands using the `-h` option of the `set` command. This makes each command name a tracked alias.

The value of all tracked aliases becomes undefined each time the `PATH` variable is reset. Another subsequent reference to the command will once again reset the alias.

11.4 Default aliases

The following aliases are compiled into the Korn shell. They may be unset or redefined at any time:

```
false='let 0'  
history='fc -l'  
integer='typeset -i'  
r='fc -e -'  
true='let 1'  
type='whence -v'  
hash='alias -t'  
functions='typeset -f'  
nohup=nohup
```

12. Shell execution options

The shell is a program like other A/UX commands, and it too has a variety of options used to control how it executes. All shell execution options can be set using the `set` command as follows:

```
set -opt[opt...]
```

Or they can be specified on the command line when you invoke a new shell or run a shell script with the `ksh` command:

```
ksh -opt[opt...] name
```

Use the `set` command to set new options in your current shell. Use the `ksh` command to invoke a subshell with the options specified or to run a script with options.

To turn options off, precede the option with a plus (+) instead of a minus (-).

The variable `$-` contains a list of all the options set. For example, if you have the `a` and `x` shell execution options set, the command

```
print $-
```

returns

```
ax
```

12.1 Options that affect the environment

- a When the `-a` shell option is set, all variable assignments result in that variable and its value being inserted in the environment. You do not need to use the `export` command to insert new values.
- k The shell execution option `-k` can be used to insert variables and values into the environment for a particular command; see “Special Environments.”

12.2 Options for Invoking new shells

In addition to the options available with the `set` command, there are four options that can be used only when a new shell is invoked with the `ksh` command.

-c *string*

If the `-c` flag is present, *string* is executed. After execution, control is returned to the parent shell. This command is often used to execute shell scripts.

-s If the `-s` flag is present or if no arguments remain, commands are read from the standard input.

-i If the `-i` flag is present, the shell is interactive. The *terminate* signal is ignored (so that `kill 0` does not kill an interactive shell), and the *interrupt* signal is caught and ignored (so that

`wait` is interruptible). In all cases, the *quit* signal is ignored by the shell.

- r If the `-r` flag is present, the shell invoked is a restricted shell. Restricted shells cannot change directories, alter the value of the `PATH` environment variable, redirect output, or specify path or command names containing the symbol `/`. See “Restricted Shells” in Chapter 2, “Bourne Shell Reference.”

13. Job control

Korn shell job control allows you to suspend current jobs, move a foreground job to the background (and vice versa), check on the status of background jobs, refer to specific background jobs by number and change their status, and receive notification when a job is done.

Every job you run in the Korn shell is associated with a **job number**; for example, when you give a background command

```
diff file1 file2 >> file3 &
```

the job number (in brackets) displays before the process ID:

```
[3] 12345
```

Job numbers are assigned sequentially, so your first job is 1, the second job is 2, and so forth.

13.1 Suspending a job

To suspend your current foreground job, type the current suspend character. Typically this is set to `CONTROL-z`, but if that does not work, you may need to set your suspend character:

```
stty susp ^z
```

(If you also intend to use shell layering, see “Using Shell Layering” on resolving possible conflicts in use of `CONTROL-z`.) Once the suspend character is set, typing it sends an immediate *stop* signal to the current job; pending output and unread input are discarded.

When the shell interprets `CONTROL-z`, it prints a message in the form

```
[job-number] + Stopped name
```

where *job-number* is the job number of the current job; `+` indicates that it is the current job; and *name* is the command name of the stopped job.

For example,

```
[2] + Stopped diff
```

13.2 Listing jobs

You can list your jobs with the command

```
jobs
```

Your jobs will be listed, and their status as running or stopped will be indicated like this:

```
[3] + Running lp chapter1 &  
[2] - Stopped vi chapter2  
[1]  Running diff file1 file2 > diff.file &
```

The + indicates the current job, and the - indicates the preceding job.

If you include the -l option, as in

```
jobs -l
```

process IDs will be shown as well as the job numbers.

13.3 Changing the status of stopped jobs

Once you have a stopped job, you can give another command at the shell prompt (leaving the job suspended), resume the job in the foreground, resume another stopped job, or continue the command processing in the background.

To leave a job suspended, do nothing. When you give the command

```
jobs
```

you will see it listed as `Stopped`. To run a stopped job in the background, give the command

```
bg %number
```

For example,

```
bg %2
```

The `bg` command with no argument

```
bg
```

puts the current (most recent) stopped job in the background to

continue executing. If a job number is given as an argument to `bg`, it must be preceded by a percent sign (%). The following notation is available for job numbers:

`%number` refers to a specific job by number
`%+` refers to the current job
`%-` refers to the preceding job
`%string` refers to the most recent stopped job that began with those characters

Thus, if you had a current stopped `lp` job whose job number was 4, you could resume this job in the background with any of the following commands:

```
bg
bg %+
bg %4
bg %lp
```

After one of these commands, you would be shown the command line of the job that was being put in the background, and then the shell prompt would be returned.

A job running in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to send output to the terminal, but this can be disabled by giving the command

```
stty tostop
```

This causes background jobs to stop when they try to send output, just as they do when they try to read input.

If a background job needs neither input nor output and completes execution in the background, the shell displays a message in the form

```
[number] + Done name
```

For example,

```
[2] + Done diff
```

You can bring a job to the foreground with the command

`fg %number`

The same conventions for referring to a stopped job given above under the `bg` command work for the `fg` command. The `fg` command works exactly like `bg`. Once your job is in the foreground, you can continue working as before.

13.4 Blocked jobs

The Korn shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked, so that no further progress is possible. For example, a job may become blocked if you execute the following sequence of commands:

```
CONTROL-Z  
bg  
fg
```

If the shell is busy with another process when it learns about a blocked job, it will wait until it is about to print another prompt before displaying a message.

13.5 Canceling jobs

To cancel a job, use the command

```
kill [%]number
```

number can be either a process ID, or a job number preceded by a percent sign (%). The rules about job numbers that apply to `bg` and `fg` also apply to the `kill` command. Using the `kill` command with process IDs to cancel jobs is discussed in “Canceling Background Commands.” Thus if you had a current background `lp` job whose job number was 4, you could cancel this job with any of the following commands:

```
kill %+  
kill %4  
kill %lp
```

The shell will display a message that the job has been terminated:

```
[4] + Terminated lp bigfile &
```

13.6 Logging out with stopped jobs

If you try to log out while any of your jobs are stopped, you will be warned with

```
You have stopped jobs.
```

If you use the `jobs` command to see what the stopped jobs are, or if you immediately try to log out again, the shell will not warn you a second time. The stopped jobs will be terminated.

The same process will occur if you attempt to log out while you have background jobs running that are not preceded by `nohup`. You will be warned once with

```
You have running jobs.
```

14. Using shell layering

Before using shell layering, you should make sure the `swtch` and `susp` characters are defined to different control sequences. Otherwise, job control will function correctly in the shell layer you invoke, but the `sh1` program will be inaccessible. The A/UX standard distribution sets `swtch` to `CONTROL-^` and `susp` to `CONTROL-z`. To check that these are defined to different control sequences on your system, enter the command

```
stty -a
```

at the shell prompt. This displays the settings for various user-definable sequences. See `stty(1)` in *A/UX Command Reference* for additional details.

For more information on the `sh1` program, see Chapter 5 “Shell Layering”.

15. Overview of shell programming

A shell program is simply a list of commands. These commands can be entered at the prompt or inserted in a file. They may contain

- variables and assignments
- typing of variables, including integer, uppercase and lowercase, justified, and so on

- one-dimensional arrays
- integer arithmetic
- control-flow statements (for example, `if`, `for`, `case`, or `while`)
- built-in shell commands
- any A/UX command

Input for the shell program can be read from the keyboard (this is the default standard input), taken from files, or embedded in the program itself (using here documents). The Korn shell also allows you to create menus that may provide input for a shell script (see “Creating and Reading a Menu”).

Shell programs can write output to the terminal screen (the default standard output), to files, or to other processes (via pipes).

When the shell program executes, each command is executed until the shell encounters either an end-of-file character or a command delimiter that directs it to stop. During execution, you can trap errors and take appropriate action.

15.1 Writing shell programs

You can enter a shell program at the prompt. When you use a built-in shell command that expects a delimiter (such as `done`) or a certain type of input, the secondary shell prompt appears after you press RETURN. This prompt (`>` by default) appears at each line until you give the expected delimiter; for example,

```
$ for i in *
> do
> cat $i
> done
$
```

Note that you can send an *interrupt* to cancel the script and return to the primary prompt.

You can also write a shell program in a text file (using a text editor), and then execute it (see “Executing Shell Scripts”). These program files are often called **shell scripts**. Note that all shell programs may be

entered at the shell prompt or inserted in a file. This does not affect their actions. Hereafter “shell scripts” will be used to refer to shell programs that reside in a file.

15.2 Executing shell scripts

There are several ways to execute a shell script; these differ mostly in terms of which instance of the shell is used for the execution.

- You can use the `ksh` command to read and execute commands contained in a file. The script will be run in a subshell, which means that it will have access only to the values set in the environment and will be unable to alter the parent shell. The command

```
ksh filename args ...
```

causes the shell to run the script contained in *filename*, taking the *args* given as positional parameters. Shell scripts run with the `ksh` command can be invoked with all the options possible for the `set` command.

- You can change the mode of the shell script file to make it executable. For example,

```
chmod +x filename
```

makes *filename* executable. Note that you may want to modify your `PATH` variable to include a personal directory (for example, `$HOME/bin`) containing your shell scripts. When you have done this, you can use your script names as ordinary commands, regardless of your current location in the file system.

Then the command

```
filename args ...
```

has the same effect as using the `ksh` command. The arguments become the positional parameters; the script is run in a subshell, which means that it will have access only to the values set in the environment and will be unable to alter the parent shell.

- You can run a shell script inside the current shell by using the “dot” command (`.`). The “dot” command (`.`) tells the current shell to run the script; no subshell is invoked. This should be

used if you wish to use local shell variables or functions, or modify the current shell:

```
. filename args ...
```

Note that there must be a space between the dot and the filename. Because the commands are executed in the current shell, this is the way to run a script that is to change values in the shell. The arguments become positional parameters. Otherwise the positional parameters are unchanged.

- You can run an executable shell script with the `exec` command. This should be used when the shell script program is an application designed to execute in place of the shell and replace interaction with it:

```
exec filename args ...
```

In this case, the shell script *replaces the current shell*. This means that when the script is over, control will not return to the shell. If you were in a login shell, you will be logged out.

15.3 Comments

A word beginning with a number sign (#) causes that word and all the following characters up to a newline to be ignored.

15.4 Writing interactive shell scripts

A shell script can invoke an interactive program such as the `vi` editor. If standard input is attached to the terminal, `vi` will read commands from the terminal and execute them just as if invoked from an interactive shell. After the session with `vi` is finished, control will pass to the next line in the script. In a similar manner, a script can invoke another copy of a shell (using `sh`, `csh`, or `ksh`), which will interpret commands from the terminal until you send an *eof*. Control will be returned to the script. You can use this to create a special environment for certain tasks by setting environment variables in a shell script and then invoking a new subshell.

You can also write interactive shell scripts by using the `read` and `eval` commands (documented below), prompting users to enter commands:

```
read command
eval $command
```

The first line will read the user's command line into the variable `command`. The `eval` command will then cause the command to execute.

15.5 Canceling a shell script

You can cancel a shell script just like an ordinary A/UX command. If the script is running in the background, use the `kill` command. See "Canceling Commands" for details on `kill` and various types of interrupts that can stop a command.

Note: Interrupts can be trapped and handled within the script using the `trap` command. See "Fault Handling and Interrupts."

15.6 Writing efficient shell scripts

In general, built-in commands execute more efficiently than A/UX commands. See "Summary of Korn Shell Commands" at the end of this chapter for a complete list of these commands. The following built-in commands are useful in constructing efficient shell scripts:

`hash` This causes the shell to remember the search path of the command named.

`ulimit` This can be used to set a limit on files written by processes.

`times` This prints the accumulated user and system times for processes.

You can also set the `-h` shell execution option using

```
set -h
```

This will locate and remember functions as they are defined, instead of when they are invoked.

Careful setting (or resetting inside a shell script) of the `PATH` and `CDPATH` environment variables make sure that the most frequently used directories are listed first. This also improves efficiency.

16. Command evaluation

When you give a command, the shell evaluates the command in one pass and then executes it. To force more than one pass of evaluation, use the `eval` command described below.

While evaluating the command, the shell performs the following substitutions on variables:

- **Alias substitution.** This checks the first word of every command to see if it is an alias, that is, a user-defined name for another command or group of commands. If an alias is found, it is replaced by the text of the alias. Only one check for aliases is made, so that an alias itself cannot contain an alias. The second alias will not be detected. For information on aliases, see “Aliases for Commonly Used Commands.”
- **Tilde substitution.** This replaces an initial tilde with a directory name (see “Shortcuts in Working With Directories”). The following forms are recognized:
 - `~` This is replaced by the value of the `HOME` variable.
 - `~name` This is replaced by the home directory of another user (where *name* is the user’s login name).
 - `~+` This is replaced by your current working directory.
 - `~-` This is replaced by your last working directory.
- **Variable substitution.** This replaces variables preceded with `$` (for example, `$user`) with their values. Only one pass of evaluation is made. For example, if the value of the variable `user` is `daphne`, then the command

```
print $user
prints
daphne
```

However, if the value of the variable `user` is `$name`, then the command

```
print $user
prints
```

\$name

The second variable is never evaluated and the value is not substituted. See “Parameter and Variable Substitution” for more information.

- **Command substitution.** The shell replaces a command enclosed in back quotes with the command’s output. For example, if the current directory is `/users/doc/virginia`, then the command

```
print `pwd`
```

prints

```
/users/doc/virginia
```

- **Blank interpretation.** The shell breaks the characters of the command line into words separated by delimiters (called “blanks”). The delimiters that are interpreted as blanks are set by the shell variable `IFS`; by default, they are blank spaces, tabs, and newlines. The null string is not regarded as a word unless it is quoted; for example,

```
print ''
```

passes the null string as the first argument to `print`, whereas the commands

```
print
```

and

```
print $null
```

(where the variable `null` is not set or set to the null string) pass no arguments to the `print` command.

- **Filename expansion.** The shell scans each word for filename expansion metacharacters (see “Using Shell Metacharacters”) and creates an alphabetical list of filenames that are matched by the pattern(s). Each filename in the list is a separate argument. Patterns that match no files are left unchanged.

These evaluations also occur in the list of words associated with a `for` loop.

16.1 Forcing more than one pass of evaluation

Sometimes more than one pass of evaluation is necessary for a command to be interpreted correctly. For example, suppose that the following two lines occur near the beginning of a shell script:

```
err_33='echo $name: user not found'  
name=elaine
```

If you give the command

```
$err_33
```

you get

```
$name: user not found
```

(which is not quite what you want). In cases like this, you can use the built-in command `eval`. So, the command

```
eval $err_33
```

forces two evaluations of the variable `err_33`. Thus, it prints

```
elaine: user not found
```

In general, the `eval` command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed.

There is an easier way to do what the above example intended, which does not require the use of `eval`. If you use double quotes (`"`), you have the following:

```
name=eli  
err_1="echo $name"
```

Then the command

```
$err_1
```

prints

```
eli
```

16.2 Command execution

After all substitution has been carried out, commands are executed as follows:

- Built-in commands, functions, and shell scripts run with the dot command (.) are executed in the current shell. The command has available all current shell execution options, the values of shell variables, environment variables, and functions defined in the current shell.
- A/UX commands, programs, executable shell scripts, shell scripts run with the ksh command, and series of commands enclosed in parentheses are executed in a subshell. The current shell invokes a child shell that executes the commands and then returns control to the parent shell. Only the values in your environment are available to these processes.
- Commands and executable scripts run with the exec command execute in place of the current shell.

If the A/UX command or program name does not specify a pathname, the environment variable PATH is used to determine which directories should be searched for the command. The only exceptions to this are built-in commands.

For more information about the execution of shell scripts, see “Executing Shell Scripts.”

16.3 Exit status: the value of the command

If a command executes successfully, in general its exit value is zero (0). If it terminates abnormally, its exit value is nonzero. The shell saves the exit value of a command. These are used primarily in shell scripts. See `signal(3)`, `exit(2)`, and `wait(2)` in *A/UX Programmer's Reference* for the values of various exit statuses.

17. Defining functions

You can use a **function definition** to assign a name to a command or list of commands. Korn shell function definitions may use the following syntax:

```
function name { command-list ; }
```

or the Bourne shell syntax:

```
name () { command-list ; }
```

In either syntax, the first brace ({) must be followed by a space or newline, and the second brace must be preceded by a semicolon or newline. See Chapter 2, “Bourne Shell Reference,” for more information about the Bourne shell syntax above.

Using the `function` keyword, a function maintaining a daily log of users could be written as follows:

```
function users { date>>log; who>>log; }
```

or

```
function users {  
date>>log; who>>log  
}
```

Note that when you use the multiline form at the shell prompt, the shell prints the secondary prompt at each line after the opening brace ({) until you enter the final brace (}).

After you have defined a function, you can use the command syntax

```
name [args]
```

For example,

```
users
```

This causes the commands in *command-list* to be executed.

Korn shell functions are read in and stored in the shell. Alias names are resolved when the function is read. Functions are executed like commands, with the arguments passed as positional parameters (see “Positional Parameters and Shell Variables”).

Functions behave like shell procedures, except for the ability to share data. Normally, the calling program and the function share variables. You can use the `typeset` command inside a function to define local variables for the function; these variables will exist only while that function (and any functions it calls) is executing.

You can cause a function to return before reaching the end of *command-list* using the command

```
return n
```

n sets the exit status of the function. If *n* is not set, the exit status is the status of the last command executed.

Functions are not typically available to an executing shell script. There are two separate ways of making a function available to an executing script. If the shell script is executing in the current shell, use the command

```
typeset -xf name
```

at the shell prompt. Functions that need to be defined across separate invocations of the shell should be defined in the `.kshrc` file (that is, the file named by the `ENV` variable).

To list the functions you have defined, use the command

```
typeset -f
```

without arguments. This displays function names and the text of functions you have entered at the keyboard.

To undefine a function, use the command

```
unset -f name
```

where *name* is the name of the function you want to remove.

18. Positional parameters and shell variables

A shell script may use two types of variables:

Positional parameters: These are string variables referred to by the numbers [0–9]. These numbers refer to the position of the parameter on the command line. Positional parameters are set on the command line and contain the arguments to the script. Positions greater than 9 must be enclosed in braces, for example, {12}, or accessed by means of `shift` (see “Changing Parameter Positions”).

Shell variables: These are string variables referred to by name. They may be assigned on the command line or inside the script itself.

The relationship between variables inside a shell script and existing shell variables depends on how the script is run. See “Executing Shell Scripts.” In all cases, shell scripts have access to the variables and values in the environment.

18.1 Positional parameters

Positional parameters may be referred to by the numbers [0-9] and set as arguments on a command line. When you enter a command at the prompt, the shell stores the elements of the command line in parameters: the command name is stored in parameter 0, the first argument is stored in parameter 1, the second argument in parameter 2, and so forth. Thus, for the command

```
diff letter1 letter2
```

parameter 0 is `diff`, parameter 1 is `letter1`, and parameter 2 is `letter2`. For the command

```
print "not a directory"
```

parameter 0 is `print` and parameter 1 is `"not a directory"`.

A shell script may refer to parameters by number; for example,

```
print $1
print $2
```

These will be substituted by the arguments given in that position on the command line; for example, for the command

```
myscript arg1 arg2
```

parameter 0 is `myscript`, parameter 1 is `arg1`, and parameter 2 is `arg2`. This prints

```
arg1
arg2
```

18.1.1 Setting values in a script

The `set` command creates a new sequence of positional parameters and assigns them values. After execution, all the old parameters are lost. For example, the command

```
set *
```

creates a sequence of positional parameters set to the names of the files in the current directory (parameter 1 is the first filename, parameter 2 is the next filename, and so on). A subsequent command,

```
set hi there
```

creates new positional parameters, discarding the old values. This time there will be only two values set; the other positional parameters will have no values. A subsequent command,

```
print $2 $1
```

displays

```
there hi
```

The command

```
print $3
```

would print a blank line, because there is no longer a parameter 3.

To set a positional parameter to a string of words separated by blanks, the entire string must be enclosed in double quotes. For example,

```
set "this is one positional parameter"
```

sets this entire string to the first positional parameter. Without the quotes, the phrase would be set, one word at a time, to the first five positional parameters.

Because the `set` command creates a new series of parameters, it is impossible to set only one parameter in a series. If only one parameter is set, it will be the first, and the remaining parameters will be lost.

The `set` command can also be used within a script to create positional parameters if none are given on the command line. Such parameters can then be used as a one-dimensional array.

After the `set` command is used to reset positional parameters, the system-maintained variable `#`, which contains the number of positional parameters, is reset to reflect the new number of parameters. For details on the system-maintained variables, see “Parameters and Variables Set by the System.”

18.1.2 Changing parameter positions

The `shift` command shifts positional parameters one or more positions to the left, discarding the value in the first position(s). The syntax is

```
shift [n]
```


If n is omitted, it defaults to 1. If n is specified, the shift takes place at the position $n+1$. For example,

```
shift 6
```

moves parameter 7 into position 1, parameter 8 into position 2, and so on, discarding the values that were stored in positions 1 through 6.

This can be useful, for example, when working through a list of files. After each file is processed, a shift can be performed, letting the next filename become parameter 1.

18.1.3 Number of parameters

The current number of positional parameters is available, stored in the system-maintained variable `#`. See “Parameter and Variable Substitution” and “Parameters and Variables Set by the System.”

18.2 Shell variables

Shell variables are named string variables. These variables can be assigned values anywhere in the script or on the command line. Variable names begin with a letter and consist of letters, digits, and underscores. Environment variables, which we have already encountered, are simply special kinds of shell variables (namely, shell variables that are available to all subshells).

18.2.1 Assigning values

Shell variables are assigned values with the syntax

```
name=value [name=value ...]
```

Note that there cannot be any spaces surrounding the equal sign.

All values are stored as strings. Pattern-matching is performed. To set a variable to a string of words separated by blanks, the entire string must be quoted; for example,

```
longvar="this is a long variable"
```

After the variable assignments

```
user="fred stone" box='???' acct=18999
```

the following values are assigned:

```
user   = fred stone
box    = ???
acct   = 18999
```

Because the Korn shell supports only string variables, all of these values (including 18999) will be strings of characters. Note that the question mark metacharacters must be quoted to prevent pattern matching, and that the value for `user` must be quoted because it contains a blank. Either single or double quotes may be used to enclose such values, provided the types are not mixed within a single value enclosure.

A variable may be set to the null string with the syntax

```
variable=
```

Shell variables may be set at the shell prompt to provide abbreviations for frequently used strings; for example,

```
b=/users/fred/bin
mv file $b
```

moves `file` from the current directory to the directory

```
/users/fred/bin
```

See “Assigning Values on the Command Line” for more information.

18.2.2 Arrays of strings

The shell supports a limited one-dimensional array facility. An element of an array parameter is referenced by a subscript, as follows:

```
variable [number]
```

number can be any arithmetic expression. The subscripts must be in the range of 0 through 511. The first subscript will be 0.

Arrays do not need to be declared. Any reference to a variable with a valid subscript is legal, and an array will be created if necessary.

The elements of an array are assigned just like individual variables; see the next section.

18.2.3 Assigning values and types to variables

Korn shell variables and arrays and array elements can be assigned in two ways:

- with an equal sign (=); for example,

```
name=diane
list[1]=first
line[10]="Please include your number"
```

- with the Korn shell `typeset` command

The `typeset` command is used

- to assign values
- to assign types
- to create constants (read-only variables)
- to export variables and functions
- to create and assign local variables within functions

This section covers using the `typeset` command to assign values, types, variables, arrays, and constants. For information on using the `typeset` command to export values to the environment, see “Customizing Your Environment.” For information on using the `typeset` command with functions, see “Defining Functions.”

The form of the `typeset` command is

```
typeset [-HLRZfilprtux[n][name[=value]]...]
```

Types may be assigned using the flag options. For *name*, you can give a variable name, the name of an array, or an indexed array element. All elements of an array must be of the same type. The *value* you give will depend on the type(s) chosen. There are no spaces around the equal sign. If no *value* is given, then *name* is simply given the type(s) specified.

The following type(s) are possible. They can be combined. If a variable (or array) that has already been assigned values changes the type from uppercase (`-u`) to lowercase (`-l`), for example, its value will usually be altered to the new type.

- H Provides A/UX-to-hostname file mapping on non-UNIX[®] machines.

- L Left justify and remove leading blanks from *value*. The width of the field remains the width assigned with the `typeset` command. When the variable is assigned a value, the value is either filled on the right with blanks or truncated as necessary to fit. Leading zeros are removed if the `-Z` option is also set. The `-L` option turns off the `-R` option. For example, you could set the width of the variable `last` to seven left-justified places as follows:

```
typeset -L last=1234567
```

or

```
typeset -L7 last
```

If `last` was then set to `Elizabeth`, which has nine characters, the last two characters (`th`) would be lost, as in the following example:

```
last=Elizabeth
print $last
Elizabe
```

If you set `last` to `Mary`, this name would be inserted in the first four places on the left and followed by three spaces.

- R Right justify and fill with leading blanks. The width of the field remains the width assigned with the `typeset` command. When the variable is assigned a value, the field is left-filled with blanks or truncated from the end as necessary to fit. This option is the reverse of the `-L` option above. The `-R` option turns off the `-L` option. Just as with the `-L` option, you can abbreviate the `-R` option (for example, `typeset -R7 last`).
- Z Used alone, or in conjunction with the `-R` option, the field is right justified and filled with leading zeros. Used in conjunction with the `-L` option, the field is left justified and any leading zeros are removed. Note that the `-Z` option does not override any `-R` or `-L` options already in use. The following examples illustrate the use of the `-Z` option with both right- and left-justified fields:

```
typeset -R5 f1 #right justify with leading blanks
f1=22
periods="....."
print "$f1"
print $periods

prints

    22
.....
```

```
typeset -Z f1 #right justify with leading zeros
print "$f1"
print $periods

prints

00022
.....
```

and

```
typeset -L f1 #left justify, remove leading zeros
print "$f1"
print $periods

prints

22
.....
```

Note: Quotation marks are necessary around the fields formatted with the `typeset` command to preserve the field interpretation you requested. If not quoted, these fields are printed without the requested justification or blank filling.

- f *names* refer to function names rather than parameter names. No assignments can be made and the only other valid option is -x. See “Defining Functions” for details.

- i The variable *name* is an integer. Declaring variables to be integers makes arithmetic done with the Korn shell `let` command much faster. A variable declared to be an integer cannot be assigned anything but an integer value. The alias

```
integer
```

is equivalent to

```
typeset -i
```

Thus,

```
integer total average
```

is the same as

```
typeset -i total average
```

The first assignment to an integer variable determines the output base. This base will be used whenever the variable is printed. The base is shown in numeric constants as

base#number

For instance, to specify that the variable `row` always be output in base two, you can define it as follows:

```
integer row=2#11010010
```

You should be sure that there are no spaces before the number sign (#); otherwise it is interpreted as the beginning of a comment. If no *base* is given, it is assumed to be 10.

- l Convert uppercase characters to lowercase. The flag `-u` is turned off.
- p Write the output of this `typeset` command, if any, to the two-way pipe created for a background command ending with “& |”. For this type of background command connected to the terminal, see “Connecting a Command to Standard Input and Output.”
- r Mark *name* read-only. Read-only variables cannot be changed while they are this type.
- t Tag the named parameters. Tags are user-definable and have no special meaning to the shell.

- u Convert lowercase characters to uppercase. The flag -l is turned off.
- x Mark *name* for automatic export to the environment. Exported parameters pass values and types to subshells but pass only values to the environment.

Using + rather than - causes certain flags to be disabled. Thus, the command

```
typeset -r OLD
```

makes the variable OLD a read-only variable, and the command

```
typeset +r OLD
```

removes this status.

Flags that may be used with + include `rxtifZRL`. Note that if a variable's only attribute is -Z, -R, or -L, use of +Z, +R, or +L will have the same effect as `unset`.

If the `typeset` command is given with options but no arguments, the variables that have these options are listed with their values. If no arguments or options are given, all variables are listed with their types.

If used inside a function, the `typeset` command creates variables local to that function. See "Defining Functions."

Use the `unset` command to remove variables.

The following is an example of the use of the `typeset` command to format data:

```
typeset -Ru10 fld1
typeset -L5 fld2
typeset -R16 fld3
typeset -LZ5 fld4
fld1="ABCdef"
fld2="002"
fld3="GHIjkl"
fld4="007"
print "$fld1 $fld2 $fld3 $fld4"
```

This sequence of commands will line up four columns of data and `print` it. In the first column will be up to ten uppercase characters, right justified; in the second column will be up to five characters, left justified; in the third column will be up to six lowercase characters, right justified; and in the fourth column will be up to five characters, left justified, with leading zeros removed. For example, if you put these commands into a file `format`, you could give the following command:

```
ksh format
```

which prints

```
ABCDEF 002   ghijkl 7
```

18.2.4 Assigning values on the command line

An argument to a shell procedure of the form `name=value`, which precedes the command name, causes `value` to be assigned to `name` before execution begins. The value of `name` in the invoking shell is not affected. For example,

```
user=fred command
```

executes `command` with `user` set to `fred`.

After variable assignments, any additional arguments are assigned to the positional parameters.

The `-k` flag causes arguments of the form `name=value` to be interpreted in this way anywhere in the argument list. See “Special Environments” for more information.

18.2.5 Removing shell variables

Shell variables are removed using the `unset` command followed by the name of the variable:

```
unset name
```

The variable and its value will be removed.

18.3 Setting constants

In the Korn shell, read-only variables whose value is intended to remain constant are declared with the command


```
typeset -r name=value
```

The variable whose *name* is given is set to *value*. Attempts to change *value* are illegal as long as the variable remains read-only. See “Assigning Values and Types to Variables” for details.

In addition, the older form,

```
readonly name ...
```

may be used.

18.4 Parameter and variable substitution

Positional parameters and shell variables are referenced and their values are substituted when the identifier (the positional parameter number or variable name) is preceded by a dollar sign (\$):

```
$identifier
```

For example,

```
$j1 $1 $8 $version
```

For variables, *identifier* can be any valid name; for positional parameters, *identifier* must be a digit between 1 and 9 inclusive, or else the *identifier* must be enclosed in braces (for example, $\${12}$).

Another notation for substitution uses braces to enclose *identifier*:

```
echo  $\{iidentifier\}$ 
```

This is equivalent to $\$iidentifier$. Braces are used when you may want to append a letter or digit to *identifier*. For example,

```
tmp=/tmp/ps  
ps a > $\{tmp\}$ a
```

substitutes the value of the variable `tmp` and directs the output of `ps` to the file `/tmp/psa`, whereas

```
ps a > $\$tmpa$ 
```

causes the value of the variable `tmpa` to be substituted.

A special shell parameter, `*`, can be used to substitute for all positional parameters (except 0, which is reserved for the name of the file being executed). The notation `@` is the same as `*` except when quoted. Thus,

```
print "$*"
```

prints all values of all the positional parameters, and

```
print "$@"
```

passes the positional parameters, unevaluated, to `print` and is equivalent to

```
print "$1" "$2" ...
```

18.5 Referencing arrays

If the variable is subscripted, the variable name and subscript must be enclosed in the braces indicated as optional above. Thus the simple variable `done` would be referenced as

```
$done
```

and the second element of the subscripted array variable `todo` would be referenced as

```
${todo[2]}
```

Referencing an array without giving a subscript is equivalent to referencing the first element, or

```
array[0]
```

because array subscripting starts with 0.

The subscript `[*]` references all the elements in an array. The number of elements in an array can be found with

```
${#array-name [*]}
```

Thus, for example, if you have the array

```
name[0]=first name[1]=second name[2]=last
```

you can give the following sequence of commands and shell responses:

```
$ print ${name[*]}  
first second last  
$ print ${#name[*]}  
3
```

18.6 Testing assignment and setting defaults

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

```
print $d
```

or

```
print ${d}
```

prints a blank line.

The following structures allow you to test whether variables or parameters are set and not null, and provide default values or messages. In these structures, *string* is evaluated only if it is to be substituted (command substitution, another variable, and so forth). If the colon is omitted, the shell checks only that the variable has been set; no action is taken if the variable or parameter is currently null.

`${identifier:-string}`

If the parameter or variable whose name is represented by *identifier* is set and is non-null, substitute its value; otherwise substitute *string*. The value of the variable or parameter is *not* changed. For example, if the variable `test` is null or unset, then

```
${test:-unset}
```

returns the string `unset`; otherwise the value of `test` is returned.

`${identifier:+string}`

If *identifier* is set and is non-null, substitute *string*; otherwise substitute nothing. The value of the variable or parameter is not changed. For example, if the variable `test` was null or unset, then

```
${test:+unset}
```

returns nothing.

`${variable:=string}`

If *variable* is not set or is null, set it to *string*; then substitute the new value. Positional parameters may not be assigned in this way. For example,

```
 ${HOME:=/user/doc}
```

tests the environmental variable `HOME` to see if it had a non-null value. If it did not, it would be assigned the value `/user/doc` and this value would be substituted. Otherwise the original value of `HOME` would be returned.

```
 ${identifier:?string}
```

If *identifier* is set and is non-null, substitute its value; otherwise print *string* and exit from the shell. If *string* is omitted, the message

```
 filename:identifier:parameter null or not set
```

prints. For example, a shell script named `distribute` that requires the parameter `directory` to be set might start as follows:

```
 echo ${directory:?}"distribution directory not set"}
```

If `directory` was not set, the script would immediately exit with the message

```
 distribute:directory:distribution directory not set
```

18.7 Creating substrings in substitution

Substrings can be created during variable substitution or they can be created with the built-in substring command. The forms of variable substitution used to create substrings are

```
 ${name#pattern} (for stripping off first characters)
```

```
 ${name%pattern} (for stripping off last characters)
```

name is the variable to be truncated. *pattern* specifies the characters to be removed. *pattern* can contain any typed characters as well as the metacharacters `*`, `?`, and `[...]`.

If *pattern* does not match any characters in the value of *name* or is null, then the original value is substituted. If *pattern* does match the beginning (with `#`) or ending (with `%`) characters, the value of *name* with the matched characters deleted is substituted. In no case is the original value of *name* changed.

For example, to substitute the filename that is the value of variable called `filename` with its extension removed, you could use the

following variable substitution:

```
 ${filename%.*}
```

18.8 Parameters and variables set by the system

Except for the question mark (?), the following variables are initially set by the shell; the ? is set by each command that executes. These variables can be referenced with the standard forms discussed above.

—	The last argument of the preceding command.
PPID	The process number of the parent of the shell.
PWD	The present working directory set by the <code>cd</code> command.
OLDPWD	The preceding working directory set by the <code>cd</code> command.
RANDOM	Each time this parameter is referenced, a random integer is generated. The sequence of random numbers can be initialized by assigning a numeric value to RANDOM.
REPLY	This parameter is set by the <code>select</code> statement and by the <code>read</code> special command when no arguments are supplied.
SECONDS	The number of seconds elapsed since login (or since the present shell was created).
?	The exit status of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise a nonzero exit status is returned. This is used in the <code>if</code> and <code>while</code> constructs for control of execution.
#	The number of positional parameters in decimal. For example, this notation is used in a script to refer to the number of arguments. An example of this use appears in the <code>case</code> section.
*	All the positional parameters (arguments) of a shell script, evaluated. For example,

```

for i in $*
do
    print $i
done

```

The above shell subroutine prints the values of all the positional parameters.

- @ Synonym for *, except when quoted. The meaning of \$* and \$@ is identical when not quoted or when used as a parameter assignment value or as a filename. When used as a command argument, however, "\$*" is equivalent to "\$1\$d\$2d...", where *d* is the first character of the IFS parameter, whereas "\$@" is equivalent to "\$1", "\$2", and so on.
- \$ The process ID of this shell in decimal. Because process numbers are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example,

```

ps a > /tmp/ps$$
command-list
rm /tmp/ps$$

```
- ! The process ID (in decimal) of the last process run in the background.
- (hyphen) The current shell flags, such as -x and -v.

19. Control-flow constructs

The shell has a variety of ways of controlling the flow of execution. The actions of the `for` loop and the `case` branch are determined by data available to the shell. The actions of the `while` or `until` loop and "if then else" branch are determined by the exit status returned by commands or tests. Control-flow constructs can be used together and loops can be nested.

In the following constructs, reserved words like `do` and `done` are recognized only following a newline or semicolon. *command-list* is a sequence of one or more simple commands separated or terminated by a newline or a semicolon.

19.1 for loops

To repeat the same set of commands for several files or arguments, use the `for` loop:

```
for name in word1 word2
do
    command-list
done
```

An example of such a procedure is `tel`, which searches a file of telephone numbers, `/usr/lib/telnos`, for the various names given as arguments to the command and passed as positional parameters.

The text of `tel` is

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

The command

```
tel fred
```

sets `i` to the name `fred` and prints those lines in the file `/usr/lib/telnos` that contain the string `fred`. It is equivalent to the form

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

The command

```
tel fred bert
```

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

To terminate a loop before the condition fails (or is met), or to continue a loop and cause it to reiterate before the end of *command-list* is reached, use the loop-control commands:

```
break [n]
continue [n]
```

These commands can appear only between the loop delimiters `do` and `done`. The `break` command terminates execution of the current loop; execution resumes after the nearest `done`. The `continue` command causes execution to resume at the beginning of the current loop.

For both `break` and `continue`, the optional *n* indicates the number of levels of enclosing loops at which execution should resume or continue. For example, the `break 2` in

```
for i in 0 1
do
  for j in 0 1
  do
    for k in 0 1 2 3
    do
      print $i$j$k
      break 2
    done
  done
done
```

causes execution to resume two levels above the current loop, printing

```
0 0 0
1 0 0
```

19.2 `select` statements

A variant form of the `for` loop is the `select` loop. Its format is

```
select identifier [in word ...] do list done
```

A `select` command prints, on standard error (file descriptor 2), the set of *words*, each preceded by a number. If *in word ...* is omitted, the positional parameters are used instead (see “Positional Parameters”). The `PS3` prompt is printed and a line is read from the standard input. If this line consists of the number of one of the listed *words*, the value of the parameter *identifier* is set to the *word* corresponding to this number. If this line is empty, the selection list is printed again. Otherwise the value of the parameter *identifier* is set to null. The contents of the line read from standard input are saved in the parameter `REPLY`. *list* is executed for each selection until a `break` or end-of-file is encountered.

The `select` command is especially useful for the generation of menus, as it sends its menu text to standard error output, leaving standard output free, so you can save replies in a file. An example of this use is given in “Creating and Reading a Menu.”

19.3 case statements

The form of the `case` statement is

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

Each *command-list* except the last must end with “; ;”. (The semicolons after the last *command-list* are optional.) This breaks out of the `case` statement after execution. After execution of *command-list*, the `case` is complete and control passes to the command following `esac`.

Patterns may include filename expansion metacharacters. However, unlike filenames, the initial dot, slashes, and a dot following a slash do not have to be matched explicitly. Different patterns to be associated with the same *command-list* are separated by the OR operator, the vertical bar (`|`). To be used literally, pattern-matching metacharacters must be quoted. Because an asterisk (`*`) matches any sequence of characters, it can be used to set up the default case. Be careful in setting up the default, however; there is no check to ensure that only one pattern matches the `case` argument. The first match found defines the set of commands to be executed. In the next example, the commands following the second pattern (`0`) will never be executed because the first pattern (`*`) executes everything it receives. The commands following the first pattern will always be executed.

```
case $# in
    *) ... ;;
    0) print "no arguments given"
       exit;;
esac
```

The following is an example of a `case` statement in a script named `append` that appends files:

```

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

```

When called with one argument, as in

```
append file
```

the system-set variable # is assigned the value 1 (the number of parameters in the call); and the standard input is appended (copied) onto the end of file using the cat command.

When called with two arguments, as in

```
append file1 file2
```

the value of # is 2 and the command appends the contents of file1 onto file2. If the number of arguments supplied to append (that is, the value of \$#) is greater than 2, then the shell prints an error message indicating proper usage.

The following example illustrates the use of alternative patterns separated by a vertical bar (|):

```

case $i in
  -x|-y) command;;
esac

```

The same effect could be had by using the bracket metacharacters ([and]), as in

```

case $i in
  -[xy]) command;;
esac

```

When using metacharacters, the usual quoting conventions apply, so that

```

case $i in
  \?) echo "input is ?" ;;
  ...
esac

```

matches the character ? for the first pattern.

A common use of the `case` construct is to distinguish among different forms of an argument. The following example is a fragment of a script that uses a `case` statement inside a `for` loop:

```
for i
do
  case $i in
    -[ocs]) ... ;;
    *) print 'unknown flag $i' ;;
    *.c) /lib/c0 $i ... ;;
    *) print 'unexpected argument $i' ;;
  esac
done
```

19.4 while loops

The `while` and `until` commands cause the program to loop depending on whether or not a certain condition is met.

A `while` loop has the form

```
while command-list1
do
  command-list2
done
```

The `while` command tests the exit status of the last simple command in *command-list1*. Each time round the loop, *command-list1* is executed. If the last command executes successfully (a zero [true] exit status is returned), then *command-list2* is executed; otherwise the loop terminates. For example, the script

```
while test $1
do
  command-list
  shift
done
```

loops through all the positional parameters. For each iteration of the loop, the `test` command is used to determine if the parameter exists. If it does, then `test` returns a zero (true) exit status and the following commands execute.

The `shift` command is used to rename the positional parameters `$2`, `$3`,... as `$1`, `$2`,..., and remove the first one, `$1`. This entire loop is equivalent to

```
for i
do
    command-list
done
```

For both `while` and `until` loops, the exit status of the loop is that of the last command executed in *command-list2*. If no commands in *command-list2* are executed, then a zero exit status is returned.

To create an endless nonconditional `while` loop, use the built-in `true` command, which always returns a zero exit status.

19.5 `until` loops

The `until` loop has the form

```
until command-list1
do
    command-list2
done
```

It works the same way as a `while` loop, except that the termination condition is reversed. Each time round the loop, *command-list1* executes; if the last command does *not* execute successfully (returns a nonzero [false] exit status), then *command-list2* is executed.

A common use for the `until` loop is to wait until some external event occurs and then run some commands. For example,

```
until test -f file
do
    sleep 300
done
command-list
```

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

To terminate a loop before the condition fails (or is met), or to continue a loop and cause it to reiterate before the end of the command list is

reached, use the loop-control commands:

```
break [n]
continue [n]
```

These commands can appear only between the loop delimiters `do` and `done`, as in the `for` loop. See “`for` Loops” for more information on using the `break` and `continue` commands.

For both `while` and `until` loops, the exit status of the loop is that of the last command executed in *command-list2*. If no commands in *command-list2* are executed, then a zero exit status is returned.

To create an endless unconditional `until` loop, use the built-in `false` command. See `true(1)` in *A/UX Command Reference* for details.

19.6 if then else

The form of the “if then else” conditional branch is

```
if command-list1
then
    command-list2
[else
    command-list3]
fi
```

In this structure, `else` and *command-list3* are optional. The `if` command tests the exit status of the last simple command in *command-list1*. If the last command executes successfully (a zero [true] exit status is returned), then *command-list2* is executed; otherwise *command-list3*, if present, is executed. For example, the `if` command can be used with the `test` command to test for the existence of a file, as below:

```
if test -f file
then
    command-list1
else
    command-list2
fi
```

Multiple conditions can be tested with a nested `if` command:

```
if condition1
then
    command-list1
else
    if condition2
    then
        command-list2
    else
        if condition3
        then command-list3
        fi
    fi
fi
```

Note that each of the nested `if` commands requires its own `fi`.
Nested ifs can also be written as

```
if condition1
then
    command-list1
elif condition2
then
    command-list2
elif condition3
then command-list3
fi
```

Note that this is a single `if` construct, with only one terminating `fi`.

An example of the `if` statement can be found in the following script.
This uses the A/UX `touch` command, which updates the last modified time for a list of files.

```

flag=
for i
do
    case $i in
    -c) flag=N ;;
    *) if test -f $i
        then
            touch $1
        elif test $flag
        then
            >$i # create it
        else
            echo "file $i does not exist"
        fi ;;
    esac
done

```

The `-c` flag in this command forces subsequent files to be created if they do not already exist. Without the `-c` flag, an error message prints if the file does not exist. The shell variable `flag` is set to some non-null string if the `-c` argument is encountered. The `ln` and `rm` commands make a link to the file and then remove it.

The exit status of the `if` command is the exit status of the last command following a `then` or `else`. If no such commands are executed, then the exit status is zero.

Conditional execution of commands can also be achieved with the symbols `&&` and `||`. See “Conditional Execution” for details.

19.7 exit

A shell script terminates when it reaches end-of-file. The exit status of the script is that of the last command executed. The built-in `exit` command can cause the script to terminate with exit status set to *n*. If *n* is omitted, exit status is that of the last command executed before `exit` was encountered.

20. Input and output

20.1 I/O redirection

All forms of input and output redirection are allowed in shell scripts. If input or output redirection (using `<` or `>`) is done in any of the control-flow commands, the entire command is executed in a subshell. This means that any values assigned during execution of the command will not be available after the command is over and control returns to the parent shell. If necessary, you can change the shell's standard input and output. See "Changing the Shell's Standard Input and Output."

20.1.1 Redirection with file descriptors

The A/UX system considers standard input, standard output, and standard error output as files, and associates a file descriptor with each of them.

File descriptors are numbers [0 to `OPEN_MAX-1`] used to identify files. By default, the file descriptors 0, 1, and 2 have the following associations:

- 0 is associated with standard input.
- 1 is associated with standard output.
- 2 is associated with standard error output.

Thus, standard input can be referenced via file descriptor 0, standard output can be referenced via file descriptor 1, and standard error can be referenced via file descriptor 2.

Input and output redirection uses the syntax

```
[x]< filename
```

and

```
[x]> filename
```

where *x* is an optional file descriptor number indicating a file; `>` and `<` are redirection operators; and *filename* is a file containing input, or to which output will be directed. The simple forms omit the file descriptor *x* and use the defaults listed above. If no descriptor appears, it is assumed to be 0 for input redirection and 1 for output redirection.

Standard error output must be redirected explicitly using either `>&` or a numeric file descriptor as documented below. The `>>` form may be used to append output to an existing file rather than overwrite the file's contents.

All file descriptors can be used with redirection characters in a command line, immediately preceding the redirection symbol. For example,

```
...2>&1 | more
```

redirects standard error on top of standard output and pipes the result through `more`.

In all forms, specifications are evaluated by the shell from left to right as they appear in the command. Filenames are subject to parameter and command substitution only. No filename expansion or blank interpretation takes place; for example, the command

```
cat testfile > *.c
```

simply writes `testfile` into a file named `“*.c”`.

20.1.2 File descriptors redirecting input

The default file descriptor for redirecting standard input is 0. This may be specified as

```
cat 0<filename
```

Because this is the default file descriptor, it may be omitted as follows:

```
cat <filename
```

20.1.3 File descriptors redirecting output

The default file descriptor for redirecting output is 1. This may be specified as

```
cat 1>filename
```

Because this is the default file descriptor, it may be omitted as follows:

```
cat >filename
```

20.1.4 Combining standard error and standard output

The default file descriptor for redirecting standard error output is 2. If you want to direct the error output of a command to a file (to save the

error messages), use the syntax

```
ls filename 2>errors
```

This saves error output (in this case, “*filename* not found”) in a file named `errors`. If you want to save the command output and error output in separate files, use the syntax

```
ls filename >output 2>errors
```

To print the output and the error output in the same file, use the syntax

```
ls filename >output 2>&1
```

This writes both standard output and error output in the file `output`. Note that `2>&1` references the `output` file because you have already redirected standard output (file descriptor 1) to this file.

For example, to save the output and the error output of the `make` command in a file named `make.log`, use the command

```
make > make.log 2>&1
```

20.1.5 Changing the shell's standard input and output

To associate standard input or standard output with a file, use the `exec` command:

```
exec >filename (for standard output)  
exec <filename (for standard input)
```

Output will be written to, or input taken from, the file specified until further redirection is done with the `exec` command. This can be useful if all output is to be taken from a file or written to a file. This construct is unlike normal shell redirection with `>` and `<` in that the redirection remains in effect until you either explicitly reset the standard I/O files, log out, or exit the current instance of the shell (shell scripts often use this feature for funny tricks).

To return output and input to the terminal, use the commands

```
exec > /dev/tty (for output)  
exec < /dev/tty (for input)
```

Reassignment can be used to avoid the problems involved in redirecting output or input in a control-flow structure.

20.1.6 Associating other files with file descriptors

The `exec` command can also be used to associate files with specific file descriptors. This can be an advantage in shell scripts that need to read or write a file line by line (see also ‘‘Reading Input’’), because writing output to a file descriptor cannot overwrite a file’s contents. The command

```
exec x<filename
```

where *x* is a number [3 to `OPEN_MAX-1`], associates *filename* with *x*. For example, the commands

```
exec 4<file1
exec 5<file2
```

associate file descriptor 4 with `file1` and file descriptor 5 with `file2`. After these commands, the syntax

```
command <&4
```

takes input from `file1`, and

```
command >&5
```

writes output to `file2`. For example,

```
$ exec 4>my.file
$ echo hello >&4
$ cat my.file
hello
$ echo bye >&4
$ cat my.file
hello
bye
```

Note that this file descriptor syntax can be repeated in a loop without overwriting the contents of `file2`.

20.2 Reading input

The built-in `read` command reads a line of input from the terminal or a file and assigns it to the variables specified. The form of the `read` command is

```
read [opt [opt] ...][name...]
```

One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. If only one *name* is specified, the entire line read will be assigned to that *name*. The exit status is zero while there is data to be read. If an end-of-file or an interrupt is encountered, the exit status is nonzero.

For example, you could use the `read` command to take input from the terminal as follows. Enter the lines

```
$read first middle last abbreviations
Alyssa Elizabeth Lynch Dr. Ph.D.
```

This would result in the following variable assignments:

```
first=Alyssa
middle=Elizabeth
last=Lynch
abbreviations=Dr. Ph.D.
```

The `read` command can also take input from a file, but will always read the first line. If you wish to move sequentially through a file, reading it line by line, you must first use the `exec` command to make the file standard input as follows:

```
exec < name.list
while read first middle last abbreviations
do
    command-list
done
exec < /dev/tty
```

In the above example, the `exec` command is used to reassign standard input to the file `name.list`. The `while` loop then uses the `read` command to read each line of the file into the variables `first`, `middle`, `last`, and `abbreviations`, and execute *command-list*. When `read` reaches the end of the file, it will return a nonzero exit status and the `while` loop will terminate. The final `exec` command then assigns standard input back to the terminal. For information about reassignment with the `exec` command, see the preceding section.

The `read` command takes the following options:

- p Take input from the input pipe of the background process connected to the parent shell with | &.
- r While reading input, \ does not indicate line continuation.
- unumber
Take input from the file whose file descriptor is given by *number*. Files and file descriptors are associated with the `exec` command. The default *number* is 0, the terminal.

The `line` command functions exactly like the `read` command, except that a whole line is read into a single variable. The line will be terminated with a newline.

20.3 Taking input from scripts

Input to a shell script can be embedded inside the script itself. This is called a **here document**. The information in a here document is enclosed as follows:

```
<<[-] word
  information
word
```

The first *word* may appear anywhere on a line; the second must appear alone and first on a line. The *words* must be identical and should not be anything that will appear in *information*. The second *word* is the end-of-file for the here document. Parameter and command substitution will occur on *information*. Normal quoting conventions apply, so that `$` can be escaped with `\`. To prevent all substitution, quote any character of the first instance of *word*. (If substitution is not required, this is more efficient.) (The type of quotes used is relevant: if *word* is single-quoted, all metacharacter expansion will be suppressed. If it is double-quoted, file, wildcard, and command substitution will take place.)

To strip leading tabs and blanks from *word* and *information*, precede the first instance of *word* with the optional hyphen (-), as follows:

```
<<-word
```

Note: If you intend to indent your code, you must use the hyphen preceding *word* unless the commands you use can

tolerate leading tabs and blanks.

For example, a shell procedure could contain the lines

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

Here the `grep` command looks for the pattern specified by `$i` in the file `/usr/lib/telnet`. This file could contain the lines

```
fred mh0123
bert mh0789
```

An alternative to using an external file would be to include this data within the shell procedure itself as a here document:

```
for i
do
    grep $i <<!
    ...
    fred mh0123
    bert mh0789
    ...
!
done
```

In this example, the shell takes the lines between `<<!` and `!` as the standard input for `grep`. The second `!` represents the end-of-file. The choice of `!` is arbitrary. Any string can be used to open and close a here document, provided that the string is quoted if white space is present and the string does not appear in the text of the here document.

Here documents are often used to provide the text for commands to be given for interactive processes, such as an editor, called in the middle of a script. For example, suppose you have a script named `change` that changes a product name in every file in a directory to a new name, as follows:

```

for i in *
do
echo $i
ed $i <<!
g/oldproduct/s//newproduct/g
w
!
done

```

(Note that `ed` commands will not tolerate leading tab characters and there is no hyphen preceding the first *word*, therefore the code is not indented.) The metacharacter `*` is expanded to match all filenames in the current directory, so the `for` loop executes once for each file. For each file, the `ed` editor is invoked. The editor commands are given in the here document between `<<!` and `!`. They direct the editor to search globally for the string `oldproduct` and each time it is found substitute the string `newproduct`. After the substitution is made, the editor saves the new copy of the file with the `w` command.

You could make the `change` script more general by using parameter substitution, as follows:

```

for i in *
do
echo $i
ed $i <<!
g/$1/s//$2/g
w
!
done

```

Now the old and new product names (or any other strings) can be given as positional parameters on the command line:

```
change string1 string2
```

Substitution of individual characters can be prevented by using a backslash (`\`) to quote the special character `$`, as in

```

for i in *
do
echo $i
ed $i <<!
1, \s/$1/$2/g
w
!
done

```

This version of the script is equivalent to the first, except that the substitution is directed to take place on the first to the last lines of the file (1, \$) instead of “globally” (g) as in the first example. This way of giving the command has the advantage that the editor will print a question mark (?) if there are no occurrences of the string \$1.

Substitution can be prevented entirely by quoting the first instance of the terminating string; for example,

```
ed $i <<\

```

Note that backslash and single quotes have the same effect in this context: all metacharacter expansion will be suppressed. Double quotes, however, will not work to prevent substitution.

To use leading tabs, precede the first *word* with a hyphen, as follows:

```

for i in *
do
    echo $i
    ed $i <<-!
        1, \s/$1/$2/g
        w
    !
done

```

20.4 Using command substitution

Command substitution can occur in all contexts where parameter substitution occurs. You can use command substitution in a shell script to avoid typing long lists of filenames. For example,

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

```


runs the `ex` editor, supplying as arguments those files whose names end in `.c` and that contain the string `TRACE`. Another example,

```
for i in `ls -t`
do
    command-list
done
```

sets the variable `i` to each consecutive filename in the current directory, with the most recent filename first.

Command substitution is also used to generate strings. For example,

```
set `date`; print $6 $2 $3, $4
```

first sets the positional parameters to the output of the `date` command and then will print; for example,

```
1986 Nov 1, 23:59:59
```

Another common example of command substitution uses the `basename` command. `basename` removes the suffix from a string so

```
basename main.c .c
```

prints the string `main`. The following fragment illustrates its application in a command substitution:

```
case $A in
...
*.c)    B=`basename $A .c`
...
esac
```

Here `B` is set to the part of `$A` with the suffix `.c` stripped off.

20.5 Writing to the standard output

The `print` command is used to write to standard output (by default, the screen). The form of the `print` command is

```
print [options] arguments [escapes]
```

The *arguments* are what is written. They are evaluated like the arguments of any other command with parameter and command substitution, filename expansion, and blank interpretation. Normal quoting conventions apply. Strings containing blanks must be enclosed

in double quotes. The arguments will be written sequentially separated by blanks, and by default they will be terminated with a newline. If there are no *arguments* or the *arguments* are unset or null variables, a blank line will be returned.

The *escapes* indicate how the *arguments* should be printed. The possible escapes are

- `\b` backspace
- `\c` print line without newline
- `\f` form feed
- `\n` newline
- `\r` carriage return
- `\t` tab
- `\v` vertical tab
- `\\` backslash
- `\n` the 8-bit character whose ASCII code is the 1-, 2-, or 3-digit octal number *n*, which must start with a zero

The backslash in each escape must be quoted; that is, it must appear twice or be enclosed in quotes. Escapes can occur anywhere among the arguments. For example, to produce two lines of output with a single `print` command, you could give the command

```
print "line one"\\n"line two"
```

You could also give the command

```
print "line one\nline two"
```

To print the value of a variable and keep the cursor on the same line, you could give the command

```
print $jj\\c
```

`print` is also useful for inserting a few lines of data into a pipe.

The *options* to the `print` command indicate how the *arguments* should be printed. These include

- This option has the same effect as no options at all and allows the first argument to begin with a dash or hyphen.
- n This option causes the output to be written without a final newline (same effect as \c).
- p This option causes the arguments to be written onto the input pipe of the background process connected to the parent shell via | &.
- r This option causes the escape sequences listed above to be ignored.
- unumber
This option causes the output to be written on the file whose file descriptor is given by *number*. Files and file descriptors are associated with the `exec` command. The default *number* is 1, the terminal.

20.6 Creating and reading a menu

The Korn shell `select` command is used to create a menu, read the response, and then execute commands (see “select Statements”). The form of the `select` command is

```
select choice in word...
do
    command-list
done
```

The `select` command first creates a menu by printing the list of *words* specified on standard error output, by default the terminal. (This is to avoid writing a menu on the output, which may be going to a file.) Each *word* is preceded by its number. The variable `PS3` is then printed below the menu as a prompt.

When the user types a response followed by RETURN, the line is read into the shell variable `REPLY` and checked to see if it corresponds to one of the menu numbers given with *words*. If `REPLY` begins with a number corresponding to a *word*, then the variable whose name is given as *choice* is set to the *word* whose number is given. Otherwise *choice* is set to null.

In any case, after the `REPLY`, *command-list* is executed. If the line typed for `REPLY` is empty, the selection menu is redisplayed.

command-list continues to be executed until a `break` or end-of-file is encountered.

For example, the commands

```
PS3="Give number of your choice "  
select activity in add delete print view stop  
do  
    case $activity in  
        add) commands;;  
        delete) commands;;  
        print) commands;;  
        view) commands;;  
        stop) break;;  
        *) print "try again";;  
    esac  
done
```

print the following on the screen:

```
1) add  
2) delete  
3) print  
4) view  
5) stop  
Give number of your choice
```

The cursor is left on the space after `choice`. When the user types the number of the activity he or she wishes, the *commands* associated with that activity in the `case` statement are executed.

For example, if the user types 2, the *commands* for `delete` are carried out. If the user types 5, for `stop`, the `select` command terminates with `break`. If the user types something not given on the menu, he or she is prompted to try again. As long as the user continues to give some `REPLY`, then after each activity as completed, the `PS3` prompt is redisplayed and he or she is given a new choice. The menu is not redisplayed.

If the user presses RETURN without specifying an activity, the menu is redisplayed along with the prompt.

Note that the final space after the string given for PS3 is necessary to avoid the user's response from following directly after the prompt.

Note: If `$activity` is replaced with `$REPLY` in the example above, the user may enter his selection as a string ("add", "delete",...) instead of a number.

21. Other features

21.1 Arithmetic evaluation

The built-in `let` command allows you to perform integer arithmetic. Evaluations are performed using long arithmetic. The form of the `let` command is

```
let expressions
```

For example, a simple `let` command could be used to increment a counter as follows:

```
let i=i+1
```

expressions will be evaluated. They can contain constants, variables, and one or more of the following operators, listed in decreasing order of precedence:

- unary minus
- ! logical negation
- * / %
multiplication, division, remainder (modulus)
- + - addition, subtraction
- <= >= < >
comparison
- == !=
equality, inequality

= arithmetic assignment

The order of precedence can be varied by enclosing subexpressions in parentheses. These will be evaluated first. The order of evaluation within a precedence group is from right to left for the = operator and from left to right for the others. The operators that have special meaning to the shell (*, <, and >) must be quoted.

Variable names must be valid identifiers. (An **identifier** is a sequence of letters, digits, or underscores, beginning with a letter or underscore.) When a variable is encountered, its value is substituted and expression evaluation resumes. Up to nine levels of recursion are permitted.

For an example of variable substitution,

```
for var in 1 2 3
do print $var
done
```

prints

```
1
2
3
```

The secondary shell prompt precedes the lines beginning with `do` and `done` when this example is entered interactively within the Korn shell.

Constants are of the form

base#number

where *base* is a decimal number between 2 and 36 representing the arithmetic base, and *number* is a number in that base. If *base* is omitted, then base 10 is used unless *number* is preceded by 0 for base 8 or 0x for base 16.

Multiple evaluations can be made with a single `let` command, as long as the expressions to be evaluated are separated by spaces. For example,

```
let average=(top+bottom)/2 "j=j*10"
```

The second expression is quoted to remove the special meaning of the character *. In addition, any individual expressions that contain spaces

must be enclosed in quotes.

The `let` command does not need to include an assignment. A standard use for the `let` command is for conditions in the `if` and `while` statements. The exit code of the `let` command is 0 if the value of the last expression is nonzero, and 1 otherwise. Thus the comparison (`<=`, `>=`, `<` and `>`) and equal operators (`==` and `!=`) can be used as follows:

```
while let "time>20"  
...
```

As long as the variable `time` has a value greater than 20, the `let` command will return an exit status of 0. When `time` is less than 20, the exit status will become 1. (For the `while` statement, see “while Loops.”)

An internal integer representation of a named variable can be specified with the `-i` option of the `typeset` special command. When this attribute is selected, the first assignment to the parameter determines the arithmetic base to be used when parameter substitution occurs.

Because many of the arithmetic operators require quoting, an alternative form of the `let` command is provided. For any command that begins with a `(`, all the characters until a matching `)` are treated as a quoted expression. More precisely,

```
((...))
```

is equivalent to

```
let "..."
```

21.2 File status and string comparison

The built-in `test` command evaluates an expression and returns a zero (true) exit status if the expression is true and a nonzero (false) exit status if the expression is false or if there is no argument. It is often used in the shell control-flow constructs.

For example,

```
test -f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. Some of the more frequently used `test` arguments are given below.

See “Summary of Korn Shell Commands” for a complete list of `test` arguments.

<code>test -L file</code>	True if <i>file</i> is a symbolic link.
<code>test file1 -nt file2</code>	True if <i>file1</i> is newer than <i>file2</i> .
<code>test file1 -ot file2</code>	True if <i>file1</i> is older than <i>file2</i> .
<code>test file1 -ef file2</code>	True if <i>file1</i> has the same device and i-node number as <i>file2</i> .
<code>test -f file</code>	True if <i>file</i> is a regular file.
<code>test -r file</code>	True if <i>file</i> is readable.
<code>test -w file</code>	True if <i>file</i> is writable.
<code>test -d file</code>	True if <i>file</i> is a directory.
<code>test s</code>	True if <i>s</i> is not the null string.
<code>test s1 = s2</code>	True if <i>s1</i> and <i>s2</i> are identical.
<code>test s1 != s2</code>	True if <i>s1</i> and <i>s2</i> are <i>not</i> identical.
<code>test n1 -eq n2</code>	True if the integers <i>n1</i> and <i>n2</i> are algebraically equal. Any of the comparisons <code>-ne</code> , <code>-gt</code> , <code>-ge</code> , <code>-lt</code> , and <code>-le</code> may be used in place of <code>-eq</code> .

In addition, there are the following operators:

- `!` the unary negation operator
- `-a` binary AND operator
- `-o` binary OR operator

The `-a` operator has higher precedence than `-o`.

All the operators and flags are separate arguments to `test`. Parentheses can be used for grouping, but must be escaped with the backslash.

A typical use of the `test` command in a shell script is the following, which prints the message “foo is a directory” if it is found to be one when the `test` command is run.


```
if test -d foo
then
    print "foo is a directory"
fi
```

People find it natural to name test files `test`, which makes it awkward (and dangerous) to use the `test` command as well. A harmless alternative is the `[]` construct, such as

```
if [ -f file ]
then
    command-list1
else
    command-list2
fi
```

The `[]` construct takes all the same arguments as `test`.

21.3 The null command (:)

The null command (`:`) does nothing and returns a zero exit status. The form of the command is

```
: args
```

The null command is therefore equivalent to the command `true`. Because it does nothing, this command can be used to introduce comments. It is generally better, however, to use the number sign (`#`) as a comment indicator, as back quotes and parentheses retain their meaning.

22. Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively.

Execution of a command may fail for any of the following reasons:

- Input/output redirection may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a bus error or memory fault signal.

- The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell will go on to execute the next command. An interactive shell will return to read another command from the terminal. If a shell script is being executed, the next command in the script will be read. Except for the last case, an error message will be printed by the shell.

All other types of errors cause the shell to exit from a shell script. Such errors include

- Syntax errors, for example, “if then done”.
- A signal such as *interrupt*. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands.

The shell flag `-e` causes the shell to terminate if an error is detected.

22.1 Fault handling and interrupts

The A/UX system uses **signals** to communicate between processes. Most signals indicate an interrupt, termination, error condition, or other break in processing. See `signal(3)` in *A/UX Programmer's Reference* for more information.

The signals that are likely to be of interest in fault handling are

- 1, hangup
- 2, interrupt
- 3, quit
- 14, alarm clock
- 15, software termination (SIGKILL sent by another process)

When a process receives a signal, it can handle it in one of three ways:

- Signals can be ignored. Some signals will cause a core dump if they are not caught.
- Signals can be caught, in which case the process must decide what action to take when the signal is received.

- Signals can be left to cause termination of the process without further action.

Note: The built-in `trap` command is only suitable for simple signal handling (for example, catching an *interrupt* from the keyboard in order to terminate the script). Functions requiring complex signal handling should be implemented as a C program. See *A/UX Programming Languages and Tools, Volume 1* for more information about the C language and associated library routines.

The built-in `trap` command allows you to detect error signals and indicate what action should be taken. The command has the form

```
trap [ command ] [ number ] ...
```

command is a command string to be read and executed when the shell receives signals whose numbers are given in *number*. *command* is scanned once when the trap is set and once when the trap is executed. `trap` commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error.

The `trap` command with *numbers* but without any arguments resets the signals whose numbers are given to their original values. If *command* is the null string, the signal whose *number* is given is ignored by the shell and by the commands it invokes. If *number* is 0, *commands* are executed on normal termination from the shell script. The `trap` command with no arguments prints a list of commands associated with each signal number.

For example,

```
trap 'rm -f /tmp/junk; exit' 2
```

sets a trap for the *interrupt* signal (2). If this signal is received, then the commands enclosed in quotes will be executed:

```
rm -f /tmp/junk; exit
```

This removes the temporary file `/tmp/junk` and then exits from the script. (`exit` is a built-in command that terminates execution of a shell procedure.) The `exit` is required; otherwise after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

The use of `trap` is illustrated in the following script:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
  case $i in
  -c)  flag=N ;;
  *)   if test -f $i
        then
          ln $i junk$$; rm junk$$
        elif test $flag
          then
            >$i
          else
            print "file '$i' does not exist"
          fi ;;
  esac
done
```

The cleanup action is to remove the file `junk$$`. (This file is named after the process ID of the script, which is kept in the system-maintained variable `$`; see “Parameters and Variables Set by the System.”) 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.

A procedure may itself elect to ignore signals by specifying the null string as the argument to `trap`. The fragment

```
trap '' 1 2 3 15
```

causes the system hangup, interrupt, quit, and software termination signals to be ignored both by the procedure and by invoked commands. These settings could be listed with the `trap` command without arguments, and reset by entering

```
trap 1 2 3 15
```

which resets the traps for the corresponding signals to their default values.

The following `scan` procedure is an example of using `trap` where there is no exit in the `trap` command:

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

This procedure steps through each directory in the current directory, prompts with its name, and then executes commands entered 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.

22.2 Debugging a shell script

Several shell options can be set that will help with debugging shell scripts. These are

- e e (error) causes the shell to exit immediately if any command exits with a nonzero exit status. (This can be dangerous in scripts involving `until` loops and other constructs where nonzero exit status is desired.)
- n n (no execute) prevents execution of subsequent commands. Commands will be evaluated but not executed. (Note that typing `set -n` at a terminal will render the terminal useless until an *eof* is entered.)

- u u (unset) causes the shell to treat unset variables as an error condition.
- v v (verbose) causes lines of the procedure to be printed as read. Use this to help isolate syntax errors.
- x x provides an execution trace. Following parameter substitution, each command is printed as it is executed.

These execution options can be turned on with the `set` command:

```
set -option
```

either inside the script or before its execution (except `-n`, which will freeze the terminal until you send an *eof*). Options may be turned off by typing

```
set +option
```

Alternatively, they can be turned on with the `ksh` command if the script is executed this way. The current setting of the shell flags is available as `$-`.

23. Summary of Korn shell commands

Input/output redirection is permitted for these commands. File descriptor 1 is the default output location. The commands

```
:  
cd  
shift
```

are treated specially as follows:

- Parameter assignment lists preceding the command remain in effect when the command completes.
- The commands are executed in a separate process when used within command substitution.

The commands

```
.  
eval  
exec  
export  
fc  
newgrp  
readonly  
return  
typeset
```

are treated specially in the following ways:

- Parameter assignment lists preceding the command remain in effect when the command completes.
- The commands are executed in a separate process when used within command substitution.
- Errors in these commands cause the script that contains them to abort.

The following is a complete summary of Korn shell built-in commands:

`: [arg...]`

The command only expands parameters. A zero exit code is returned. This is equivalent to `true`, so that `while :` is equivalent to `while true`. For example,

```
while : `echo hi > /dev/tty`  
do  
...  
done
```

Note that expressions in back quotes or parentheses may have side effects when used as arguments. See “while Loops.”

`. file [arg...]`

Read and execute commands from *file* and return. The commands are executed in the current shell environment. The search path specified by `PATH` is used to find the directory containing *file*. If any arguments are given, they become the positional parameters. Note that this differs from `sh(1)`. Otherwise, the positional parameters are unchanged. See

“Executing Shell Scripts.”

```
alias [-tx][name[=value]]...
```

With no arguments, print the list of aliases in the form *name=value* on standard output. An alias is defined for each *name* whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution.

value may not contain an alias. For example, if you set the following aliases:

```
alias l = ls
alias ll = l
```

then the command

```
l
```

is translated into `ls` and a listing of files appears. If, however, you give the command

```
ll
```

this prints

```
ksh: l: not found
```

as no further translation (or alias evaluation) takes place.

Note: Aliases may not contain aliases.

The `-t` flag is used to set and list tracked aliases. The value of a tracked alias is the full pathname corresponding to the given *name*. *value* becomes undefined when the value of `PATH` is reset, but the aliases remained tracked. Without the `-t` flag, for each *name* in the argument list for which no *value* is given, the name and value of the alias is printed.

The `-x` flag is used to set or print exported aliases. An exported alias is defined across subshell environments. `alias` returns true unless a *name* is given for which no alias has been defined. See “Defining an Alias.”

`bg [%job]`

If *job* is specified, put it into the background; otherwise put the current job in the background. See “Job Control.”

`break [n]`

Exit from the enclosing `for`, `while`, `until`, or `select` loop, if any. If *n* is specified, break *n* levels. See “for Loops.”

`cd [arg]`

`cd old new`

This command can be in either of two forms. In the first form, it changes the current directory to *arg*. If *arg* is `--`, the directory is changed to the previous directory. The shell parameter `HOME` is the default *arg*. The parameter `PWD` is set to the current directory. The shell parameter `CDPATH` defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (`:`). The default path is `<null>` (specifying the current directory). Note that the current directory is specified by a null pathname, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with `/`, the search path is not used. Otherwise each directory in the path is searched for *arg*.

The second form of `cd` substitutes the string *new* for the string *old* in the current directory name, `PWD`, and tries to change to this new directory.

See “Shortcuts in Working With Directories.”

`continue [n]`

Resume the next iteration of the enclosing `for`, `while`, `until`, or `select` loop. If *n* is specified, resume at the *n*th enclosing loop. See “for Loops.”

`echo [-n][arg...]`

The built-in `echo` command writes its arguments (separated by blanks and terminated by a `RETURN`) on the standard output (see also `print`). If the `-n` flag is used, no newline is added to the output. `echo` is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send

diagnostics to the standard error file, do

```
echo ... 1>&2
```

`eval [arg...]`

Read arguments as input to the shell and execute the resulting commands. See “Command Evaluation.”

`exec [arg...]`

If *arg* is given, execute the command specified by the arguments in place of this shell without creating a new process. Input/output arguments may appear and affect the current process. If no arguments are given, the effect of this command is to modify file descriptors as prescribed by the input/output redirection list. In this case, any file descriptor numbers greater than 2 that are opened with this mechanism are closed when invoking another program. See “Executing Shell Scripts.”

`exit [n]`

Cause the shell to exit with the exit status specified by *n*. If *n* is omitted, the exit status is that of the last command executed. An *eof* will also cause the shell to exit, unless the shell has the `ignoreeof` option turned on (see `set`). See “Fault Handling and Interrupts.”

`export [name...]`

Mark *names* for automatic export to the environment of subsequently executed commands. See “The Environment.”

`fc [-e ename][-nlr][first][last]`

`fc -e -[old=new][command]`

In the first form, a range of commands from *first* to *last* is selected from the last `HISTSIZE` commands that were typed at the terminal. The arguments *first* and *last* may be specified as a number or as a string. A string is used to locate the most recent command starting with the given string. A negative number is used as an offset to the current command number. If the flag `-l` is selected, the commands are listed on standard output. Otherwise the editor program *ename* is invoked on a file containing these keyboard commands. If *ename* is not supplied, the value of the parameter `FCEDIT` (default `/bin/ed`) is used

as the editor. When editing is complete, the edited commands are executed. If *last* is not specified, it will be set to *first*. If *first* is not specified, the default is the preceding command for editing and -16 for listing. The flag -r reverses the order of the commands, and the flag -n suppresses command numbers when listing.

In the second form, the *command* is reexecuted after the substitution *old=new* is performed. See “Editing and Reusing Commands.”

fg [%*job*]

If *job* is specified, bring it to the foreground; otherwise bring the current job into the foreground. See “Job Control.”

hash

This causes the shell to remember the search path of the command named. See “Writing Efficient Shell Scripts.”

jobs [-l]

List the active jobs. Given the -l option, list process IDs in addition to the normal information. See “Job Control.”

kill [-sig] *process* ...

Send either the terminate signal or a specified signal to the specified jobs or processes. Signals are given either by number or by name (as given in `signal(3)` in *A/UX Programmer's Reference* stripped of the prefix SIG). The signal numbers and names can be listed by typing

```
kill -l
```

If the signal being sent is SIGTERM or SIGHUP, the job or process will be sent a continue signal if it is stopped. *process* can be either a process ID or a job number. See “Canceling a Background Command” and “Job Control.”

let *arg* ...

Each *arg* is an arithmetic expression to be evaluated. All calculations are done as long integers and no check for overflow is performed. Expressions consist of constants, named parameters, and operators. The following set of operators, listed in order of precedence, has been implemented:

- unary minus
- ! logical negation
- * / %
multiplication, division, remainder (modulus)
- + - addition, subtraction
- <= >= < >
comparison
- == !=
equality, inequality
- = arithmetic assignment

Subexpressions in parentheses, (), are evaluated first and can be used to override the above precedence rules. The evaluation within a precedence group is from right to left for the = operator and from left to right for the others.

A parameter name must be a valid identifier. When a parameter is encountered, the value associated with the parameter name is substituted and expression evaluation resumes. Up to nine levels of recursion are permitted.

The return code is 0 if the value of the last expression is nonzero, and 1 otherwise. See “Arithmetic Evaluation.”

`newgrp [arg...]`
Equivalent to

`exec newgrp arg...`

See `newgrp(1)` in *A/UX Command Reference*.

`print [-Rnprsu[n]] [arg...]`

The shell output mechanism. With no flags or with flag -, the arguments are printed on standard output as described by `echo`. In raw mode, -R or -r, the escape conventions of `echo` are ignored. The -R option will print all subsequent arguments and options other than -n. The -p option causes the arguments to be written onto the pipe of the process spawned with | & instead of

standard output. The `-s` option causes the arguments to be written onto the history file instead of standard output. The `-u` flag can be used to specify a one-digit file descriptor unit number *n* on which the output will be placed. The default is 1. If the flag `-n` is used, no newline is added to the output.

`pwd`

Print the current working directory. This is equivalent to

```
print -r -$PWD
```

`read [-prsu[n]][name?prompt][name...]`

The shell input mechanism. One line is read and broken up into words using the characters in `IFS` as separators.

In raw mode, `-r`, a `\` at the end of a line does not signify line continuation. The first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*.

The `-p` option causes the input line to be taken from the input pipe of a process spawned by the shell using `|&`. If the `-s` flag is present, the input will be saved as a command in the history file. The flag `-u` can be used to specify a one-digit file descriptor unit to read from. The file descriptor can be opened with the `exec` special command.

The default value of *n* is 0. If *name* is omitted, `REPLY` is used as the default *name*. The return code is 0 unless an end-of-file is encountered. An end-of-file with the `-p` option causes cleanup for this process so that another can be spawned. If the first argument contains a `?`, the remainder of this word is used as *prompt* when the shell is interactive. If the given file descriptor is open for writing and is a terminal device, *prompt* is placed on this unit. Otherwise *prompt* is issued on file descriptor 2. The return code is 0 unless an end-of-file is encountered. See “Reading Input.”

`readonly [name...]`

Mark the given *names* read-only. These names cannot be changed by subsequent assignment. See “Setting Constants.”

`return` [*n*]

Cause a shell function to return to the invoking script with the return status specified by *n*. If *n* is omitted, the return status is that of the last command executed. If `return` is invoked while not in a function or a script, it is the same as `exit`. See “Defining Functions.”

`set` [-aefhkmnopstuvx][*-o option...*][*arg...*]

- a Automatically export all subsequent parameters that are defined.
- e If the shell is noninteractive and if a command fails, execute the ERR trap, if set, and exit immediately. This mode is disabled while reading profiles.
- f Disable filename generation.
- h Each command whose name is an identifier becomes a tracked alias when first encountered.
- k Place all parameter assignment arguments in the environment for a command, not just those that precede the command name.
- m Run background jobs in a separate process group and print a line upon completion. The exit status of background jobs is reported in a completion message. On systems with job control, this flag is turned on automatically for interactive shells.
- n Read commands but do not execute them. Ignored for interactive shells.
- o The following arguments can be one of the following option names:
 - `allexport` Same as `-a`.
 - `errexit` Same as `-e`.
 - `bgnice` All background jobs are run at a lower priority.

<code>ignoreeof</code>	The shell will not exit on <i>eof</i> . The <i>exit</i> command must be used.
<code>keyword</code>	Same as <code>-k</code> .
<code>markdirs</code>	All directory names resulting from filename generation have a trailing <code>/</code> appended.
<code>monitor</code>	Same as <code>-m</code> .
<code>noexec</code>	Same as <code>-n</code> .
<code>noglob</code>	Same as <code>-f</code> .
<code>nounset</code>	Same as <code>-u</code> .
<code>protected</code>	Same as <code>-p</code> .
<code>verbose</code>	Same as <code>-v</code> .
<code>trackall</code>	Same as <code>-h</code> .
<code>vi</code>	Puts you in insert mode of a <i>vi</i> -style in-line editor until you press <code>ESCAPE</code> , which puts you in move mode. A <code>RETURN</code> sends the line.
<code>viraw</code>	Each character is processed as it is typed in <i>vi</i> mode.
<code>xtrace</code>	Same as <code>-x</code> .

If no option name is supplied, the current option settings are printed.

- `-p` Reset the `PATH` variable to the default value, disable processing of the `$HOME/.profile` file, and use the file `/etc/suid_profile` instead of the `ENV` file. This mode is automatically enabled whenever the effective user ID (group ID) is not equal to the real user ID (group ID).
- `-s` Sort the positional parameters.
- `-t` Exit after reading and executing one command.

- u Treat unset parameters as an error when substituting.
- v Print shell input lines as they are read.
- x Print commands and their arguments as they are executed.
- Turn off -x and -v flags and stop examining arguments for flags.
- Do not change any of the flags. This is useful in setting \$1 to a value beginning with -. If no arguments follow this flag, the positional parameters are unset.

Using + rather than - causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in \$-. The remaining arguments are positional parameters and are assigned, in order,

\$1 \$2 ...

If no arguments are given, the values of all names are printed on the standard output. See “The Environment.”

`shift [n]`

Rename the positional parameters from

\$n+1 ...

to

\$1 ...

The default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to \$#. See “Changing Parameter Positions.”

`test [expr]`

Evaluate conditional expression *expr*. See “File Status and String Comparison.” The arithmetic comparison operators are not restricted to integers. They allow any arithmetic expression. Four additional primitive expressions are allowed:

- L *file*
true if *file* is a symbolic link

file1 -nt *file2*
true if *file1* is newer than *file2*

file1 -ot *file2*
true if *file1* is older than *file2*

file1 -ef *file2*
true if *file1* has the same device and i-node number as *file2*

Note that the left bracket, [, is a synonym for `test`, but must be matched by a right bracket,].

See “File Status and String Comparison.”

`times`

Print the accumulated user and system times for the shell and for processes run from the shell. See “Writing Efficient Shell Scripts.”

`trap [arg][sig] ...`

arg is a command to be read and executed when the shell receives signal(s) *sig*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Each *sig* can be given as a number or as the name of the signal. `trap` commands are executed in order by signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective.

If *arg* is omitted or is `-`, then all *sigs* are reset to their original values. If *arg* is the null string, this signal is ignored by the shell and by the commands it invokes.

If *sig* is `ERR`, *arg* will be executed whenever a command has a nonzero exit code. This trap is not inherited by functions. If *sig* is `0` or `EXIT` and the `trap` statement is executed inside the body of a function, the command *arg* is executed after the function completes. If *sig* is `0` or `EXIT` for a trap set outside any function, the command *arg* is executed on exit from the shell. The `trap` command with no arguments prints a list of commands associated with each signal number. See “Fault Handling and Interrupts.”

`typeset [-HLRZfilprtux][n][name[=value]]...`

When invoked inside a function, create a new instance of the parameter *name*. The parameter value and type are restored when the function completes. The following attributes may be specified:

- H Provide A/UX-to-hostname file mapping on non-A/UX machines.
- L Left justify and remove leading blanks from *value*. If *n* is nonzero, it defines the width of the field; otherwise the width is determined by the width of the value of the first assignment. When the parameter is assigned *value*, it is filled on the right with blanks or truncated if necessary to fit into the field. Leading zeros are removed if the `-z` flag is also set. The `-R` and/or `-z` flags are turned off.
- R Right justify and fill with leading blanks. If *n* is nonzero, it defines the width of the field; otherwise the width is determined by the width of the value of the first assignment. The field is left filled with blanks or truncated from the end if the parameter is reassigned. The `-L` flag is turned off.
- Z Right justify and fill with leading zeros if the first nonblank character is a digit and the `-L` flag has not been set. Used in conjunction with the `-L` option, the field is left justified and any leading zeros are removed. If *n* is nonzero, it defines the width of the field; otherwise the width is determined by the width of the value of the first assignment.
- f *name* refers to function name rather than parameter name. No assignments can be made, and the only other valid flags are `-t`, which turns on execution tracing for this function, and `-x`, which allows the function to remain in effect across shell procedures executed in the same process environment.
- i Make parameter an integer. This makes arithmetic faster. If *n* is nonzero, it defines the output arithmetic base; otherwise the first assignment determines the output base.

- l Convert all uppercase characters to lowercase. The uppercase flag, -u, is turned off.
- p Write output of the command, if any, to the two-way pipe.
- r Mark the given *names* read-only. These names cannot be changed by subsequent assignment.
- t Tag the named parameters. Tags are user definable and have no special meaning to the shell.
- u Convert all lowercase characters to uppercase. The lowercase flag, -l, is turned off.
- x Mark the given *names* for automatic export to the environment of subsequently executed commands.

Using + rather than - causes these flags to be turned off. If no *name* arguments are given but flags are specified, the `typeset` command prints a list of *names* (and optionally *values*) of the parameters that have these flags set. (Using + rather than - keeps the values to be printed.) If no *names* and flags are given, the names and attributes of all parameters are printed. See “Assigning Values and Types to Variables.”

`ulimit [-f][n]`

- f Impose a size limit of *n* 512-byte blocks on files written by child processes (files of any size may be read). If no option is given, -f is assumed. If *n* is not given, the current limit is printed.

See “Writing Efficient Shell Scripts.”

`umask [nnn]`

Set the user file-creation mask *nnn*. (See `umask(2)` in *A/UX Programmer's Reference*). If *nnn* is omitted, the current value of the mask is printed.

`unalias name ...`

Remove the parameters given by the list of *names* from the alias list. See “Listing and Removing Aliases.”

`unset [-f] name ...`

The parameters given by the *names* are unassigned; that is, their

values and attributes are erased. Read-only variables cannot be unset. If the flag `-f` is set, the names refer to function names. See “Removing Shell Variables.”

`wait [n]`

Wait for the specified child process and report its termination status. If *n* is not given, all currently active child processes are waited for. The return code from this command is that of the process waited for. (See `wait(2)` in *A/UX Programmer's Reference*.)

`whence [-v] name ...`

For each *name*, indicate how it would be interpreted if used as a command name. The flag `-v` produces a more verbose report. See “Learning About Built-in Commands.”

Chapter 4

C Shell Reference

Contents

1. The C shell prompt	1
1.1 The secondary shell prompt	1
1.2 Changing the prompt character	1
2. Types of commands	1
3. The parts of a command	2
4. Interactive use	3
4.1 Command termination character	3
4.2 Impossible commands	3
4.3 Background commands	4
4.3.1 Checking command status	4
4.3.2 Logging out	4
4.4 Canceling commands	5
4.4.1 Before you press RETURN	5
4.4.2 While a command is running	5
4.4.3 Canceling background commands	7
5. Listing and reusing commands	7
5.1 Listing previous commands	8
5.2 Reusing a previous command	8
5.3 Changing text in the most recent command line	9
5.4 Editing and reexecuting previous commands	10
5.5 Reusing parts of previous command lines	12
5.6 Using modifiers with your command history	13
5.7 Other uses for command history	15
6. Using shell metacharacters	16
6.1 Specifying home directories	18

6.2	Specifying filenames with metacharacters	18
6.3	I/O redirection	21
6.4	Combining commands: pipelines	23
6.5	Command grouping	24
6.6	Conditional execution	25
6.7	Quoting	25
7.	Working with more than one shell	27
7.1	Changing to a new shell	28
7.2	Changing your default shell	28
8.	The environment	28
8.1	Global environment variables	29
8.1.1	Listing existing values	29
8.1.2	Adding environment variables and modifying values	30
8.1.3	Removing environment variables	30
8.1.4	Commonly used environment variables	30
8.2	C shell variables	32
8.2.1	Listing existing values	32
8.2.2	Adding C shell variables and modifying values	32
8.2.3	Removing C shell variables	33
8.2.4	C shell variables	33
8.3	The environment and new shell instances	37
8.4	Special environments	37
8.5	The default environment on your system	38
9.	The <code>.login</code> file	38
9.1	A sample <code>.login</code> file	39
9.1.1	Locating commands	39
9.1.2	Your editing environment	40
9.2	Customizing your login procedure	40
10.	The <code>.cshrc</code> file	41
10.1	A sample <code>.cshrc</code> file	41
10.1.1	Using history numbers as your prompt	41
10.1.2	Protection against unintentional logout	42

11. Aliases for commonly used commands	42
11.1 Defining an alias	42
11.2 Listing and removing aliases	44
11.3 Aliases that take arguments	44
12. Shell execution options	45
13. Job control	46
13.1 Suspending a job	47
13.2 Listing jobs	47
13.3 Changing the status of stopped jobs	47
13.4 Blocked jobs	49
13.5 Canceling jobs	49
13.6 Logging out with stopped jobs	50
14. Using shell layering	50
15. Overview of shell programming	51
15.1 Writing shell programs	51
15.2 Executing shell scripts	52
15.3 Comments	53
15.4 Writing interactive shell scripts	53
15.5 Canceling a shell script	53
15.6 Writing efficient shell scripts	54
16. Command evaluation	54
16.1 Command execution	56
16.2 Exit status: the value of the command	57
17. Arguments and shell variables	57
17.1 Arguments	58
17.2 Shell variables	58
17.2.1 Assigning values	59
17.2.2 Changing position of elements	60
17.2.3 Removing shell variables	60
17.3 Variable substitution	61
17.4 Testing assignment	64
17.5 Variables set by the system	64
18. Control-flow constructs	65
18.1 foreach loops	65

18.2	switch statements	66
18.3	while loops	66
18.4	if then else	67
18.5	goto	67
18.6	exit	68
19.	Input and output	68
19.1	Standard error and output files	68
19.2	Reading input	69
19.3	Taking input from scripts	69
19.4	Using command substitution	72
19.5	Writing to the standard output	73
20.	Other features	73
20.1	Arithmetic evaluation	73
20.2	Expressions	74
20.3	File status	75
21.	Error handling	76
21.1	Fault handling and interrupts	76
21.2	Debugging a shell script	77
22.	Summary of C shell commands	77

Chapter 4

C Shell Reference

1. The C shell prompt

The C shell is a program that interprets commands and arranges for their execution. The C shell displays a character called the **prompt** (or **primary shell prompt**) whenever it is ready to begin reading a new command from the terminal. By default, the C shell prompt character is set to the percent sign (%).

1.1 The secondary shell prompt

If you press the RETURN key when the shell expects further input, you will see the **secondary shell prompt**. By default, this prompt character is set to the question mark (?). Like the primary shell prompt, this can be redefined.

When you enter a multiline construct (such as a `foreach` loop) at the shell prompt, the question mark appears as the first character of each line until you give the final delimiter. When you see a ? as a prompt (either because you are using a multiline construct or because of an error), an *interrupt* will stop the process and issue the primary shell prompt (%) for another command. See “Canceling Commands” for information about the *interrupt* on your system.

1.2 Changing the prompt character

You may change the primary shell prompt character by redefining the local environment variable `prompt` to any other character or string of characters. See “C Shell Variables.”

2. Types of commands

The shell works with three types of commands:

Built-in shell commands: Built-in commands are written into the shell itself and are generally used for writing shell programs. Each A/UX shell has a slightly different set of built-in commands. The built-in C shell commands are listed under “Summary of C Shell

Commands.”

A/UX commands: Every shell can also invoke all A/UX commands (see “Command Summary by Function” in *A/UX Command Reference* for a complete list of these). A/UX commands are executable programs stored in system directories such as `/bin` and `/usr/bin`. When you enter an A/UX command (for example, `ls`), the shell searches all directories specified by your `PATH` variable (see “Locating Commands”) to locate the program and invoke it.

User-defined commands: You can combine built-in shell commands and A/UX commands to define your own **shell programs** (see “Overview of Shell Programming”). Shell programs can be typed in at the shell prompt or entered in a file. A shell program contained in a file is generally called a **shell script**. Once a shell script is defined, with certain limitations, it can be used like any other command or program.

You can also create your own commands using a high-level language such as C. See *A/UX Programming Languages and Tools, Volume 1* for more information.

3. The parts of a command

Whenever you see a shell prompt, you can enter a command by typing the command name. Most A/UX commands have one or more **flag options**, which can follow the command name to modify the way the command operates. Flag options are usually a hyphen followed by one or more characters; for example, `-l` is a flag option to the `ls` command:

```
ls -l
```

In this case, the `-l` is a flag option that modifies the way the `ls` command operates, producing a “long” listing that contains more information than the standard `ls` output. For the flag options that apply to a particular A/UX command, see the manual page entry for that command in *A/UX Command Reference*. For options to the C shell built-in commands, see “Summary of C Shell Commands.”

Many A/UX commands also expect one or more **arguments**, which pass information to the command. An argument may be any parameter

expected by the command; for example, a directory name may be an argument to the `ls` command:

```
ls /bin
```

In this case, the directory name `/bin` is an argument that specifies which directory the `ls` command should list.

The entire command specification, including any options and arguments, is called the **command line**. A command line is terminated by RETURN. For example, in the command line

```
ls -l /bin
```

`ls` is the command name, `-l` is a flag option (specifying a “long” listing), and `/bin` is an argument (specifying which directory to list).

To give a command longer than one line, you must precede the final RETURN with a backslash (`\`). This prevents the shell from interpreting RETURN as the end of a command. You can continue this for several lines; the shell will wait for a plain RETURN (not preceded by a backslash) to execute the multiline command.

Commands can also be combined; see “Command Grouping.”

4. Interactive use

4.1 Command termination character

When you are entering commands to the shell interactively, the shell will not begin executing the command until you press the RETURN key. Therefore, if you mistype something, you can backspace and correct the mistake before pressing RETURN. When the shell recognizes the RETURN, it executes the command line; when the process completes, a new prompt will be printed on the screen. The shell is now ready to accept further commands.

4.2 Impossible commands

If you give an impossible command (a command or command line that doesn’t exist or uses improper syntax), the shell will print an error message and return the prompt for another command.

4.3 Background commands

You can direct the shell to execute commands in the “background” while you continue to work at the shell prompt (the “foreground”). To run background processes, end the command line with an ampersand (&) before the final RETURN. For example,

```
cat file1 file2 > bigfile &  
[1] 1234
```

The number shown in brackets below the command line is the job number; the other number is the **process ID** (PID) associated with the sample `cat` command as long as it is executing. After the process ID is displayed, the shell returns the prompt so you can use the terminal immediately for other work.

Note: To save the output from a job you are running in the background, you must redirect it into a file or pipe it to a printer. If you do not redirect the command output, it will appear on your screen and will not be saved. In addition, remember that the output of a background command is not complete until the command has finished. The presence of a prompt does not mean that the output is ready for use.

To suspend processes that require input from the keyboard (such as an editor or a remote login across a network), you can simply send a *suspend* to temporarily stop the job. See “Job Control” for more information.

4.3.1 Checking command status

To check on the status of a background command, use

```
jobs
```

This command shows the **process status** of all your commands; they are identified by job number, process ID, and by name. See “Job Control” for more details.

4.3.2 Logging out

If you are logged out of the system while running a foreground job (for example, if a telephone connection is lost or the `getty` process on

your terminal is disconnected), the shell terminates your foreground processes. You can prevent this by using the `nohup` command (which stands for “no hang up”) as follows:

`nohup command`

This also applies if you stop a foreground job using a *suspend* signal, and then log out. If you ran the foreground job with `nohup`, the job will remain (stopped) after you log out.

If you are running a job in the background, you do not need to use `nohup`; your background process will continue to run after you log out (see “Background Commands”). See `nohup(1)` in *A/UX Command Reference* for details.

4.4 Canceling commands

A number of special control sequences come into play when canceling commands. The A/UX standard distribution defines these sequences as follows:

Name	A/UX standard distribution
<i>interrupt</i>	CONTROL-c
<i>quit</i>	CONTROL-l
<i>erase</i>	DELETE
<i>kill</i>	CONTROL-u
<i>eof</i>	CONTROL-d
<i>swtch</i>	CONTROL-`
<i>susp</i>	CONTROL-z

However, you may reassign any of these sequences using the `stty` command. See `stty(1)` in *A/UX Command Reference* for more information.

4.4.1 Before you press RETURN

If you type part of a command and then decide you do not want to execute it, you can send an *interrupt* or *kill* to the system at any point in the command line.

4.4.2 While a command is running

There are several ways to stop a command that is executing:

- Send the *interrupt* signal.

For example, the output of a command such as

```
cat /etc/termcap
```

will scroll by on your terminal. If you want to terminate the process, you can send the *interrupt* signal. Because the `cat` command does not take any precautions to avoid or otherwise handle this signal, the *interrupt* will cause it to terminate.

- Use CONTROL-s to suspend scrolling output.

The A/UX control-flow keys are CONTROL-s (suspend scrolling output) and CONTROL-q (resume scrolling output). You can use these to stop a screenful of output, resume scrolling, and stop a screenful again. CONTROL-s and CONTROL-q cannot be redefined using `stty`; however, `stty` can enable and disable control-flow.

- Send an *eof* character.

Many programs (including the shell) terminate when they get an *eof* from their standard input. You could accidentally terminate the shell (which would log you off the system) if you enter *eof* at a prompt or, in terminating some other program, if you send an *eof* one time too many. See “C Shell Variables” for information about the `ignoreeof` option; when this option is set, the shell will not terminate when it receives an *eof*.

- Wait for the end-of-file condition from a file.

If a command has its standard input redirected from a file, then it will terminate normally when it reaches the end of that file. If you give the command

```
mail ellen < note
```

(where `note` is an existing file), the `mail` program will terminate when it detects the end-of-file condition from the file.

- Send the *quit* signal.

If you run programs that are not fully debugged, it may be

necessary to stop them abruptly. You can stop programs that hang or repeat inappropriately by using *quit*. This will usually produce a message such as

```
Quit (Core dumped)
```

indicating that a file named `core` has been created containing information about the state of the running program when it terminated because of the *quit* signal. You can examine this file yourself, or forward information to the person who maintains the program telling him or her where the `core` file is.

- Send a *suspend* signal.

You can send a *suspend* signal to temporarily stop commands that are executing. You can then resume the job or cause it to run in the background. See “Job Control” for more information.

4.4.3 Canceling background commands

If you have a job running in the background and decide you do not want the command to finish executing, use the `kill` command.

When a job is running in the background, it ignores *interrupt* and *break* signals. To terminate a background command, use

```
kill process-ID
```

The `kill` command takes as an argument the process ID or the job number preceded by a percent sign (%). See “Job Control” and “Summary of C Shell Commands” for information on the `kill` command.

5. Listing and reusing commands

The C shell retains your most recent *n* commands in accordance with the setting of the `history` variable. In the `/etc/cshrc` file in the A/UX standard distribution, this variable is set to 200. You can change the number of commands the shell remembers by setting the `history` variable to another number. See “C Shell Variables” for more information.

The exclamation mark (!) invokes the C shell history substitution mechanism. The ! may be preceded by the \ escape character to

prevent it from being interpreted with this special meaning.

History substitution allows you to reexecute previous commands or reuse words from a previous command as portions of a new command. History substitutions begin with the `!` character and may begin anywhere in the command line. (Note, however, that you cannot nest history substitutions by using more than one `!` character on a command line.)

History substitutions also occur when an input line begins with the caret (`^`). See “Changing Text in the Most Recent Command Line.”

5.1 Listing previous commands

To see the list of your previous commands, type

```
history
```

This prints a numbered list of commands, from your 50th (or *n*th) previous command to your most recent. For example,

```
101 mail
102 vi note
103 mail ellen < note
104 date
105 ls
106 cd revisions/additions
107 ls
108 vi prog.c
109 wc prog.c
110 cd /usr/source/information
111 history
```

5.2 Reusing a previous command

The exclamation mark (`!`) character is used to reexecute previous commands. To reexecute your most recent command, use the command

```
!!
```

This will echo the previous command line on the screen and reexecute it.

The `!` character can also be followed by a command number or a string that identifies the beginning of a previous command line. For example, the command

```
!108
```

echoes and reexecutes command number 108 from the list above.

You may also reuse a command by specifying a string that identifies it; for example, in the history list above, the command

```
!v
```

echoes and invokes

```
vi prog.c
```

The exclamation mark (`!`) may be used in the following notations:

- !! Repeat the most recent command.
- !*n* Repeat the *n*th command, where *n* is the (history) number of a previous command. This is illustrated above.
- !*s* Repeat the most recent command beginning with the string *s*. *s* is one or more characters. For example, you could repeat the `cd` command number 106 by typing

```
!cd
```
- !*-n* Repeat the command that occurred *n* commands preceding this command line (current line *-n*).
- !*?s?* Repeat the most recent command that contains the string *s* anywhere in the command line.

5.3 Changing text in the most recent command line

You may also edit previous command lines. In the simplest case, where you are modifying the text of the most recent command, use the shorthand notation

```
^old^new^
```

This is useful for correcting typing errors in a long command (where *old* identifies the typing error and *new* is the corrected spelling) or for modifying the most recent command to run with a different parameter

(such as filename).

Note: The caret (^) shorthand only works on the most recent command. It must be used on the command line that immediately follows the command you wish to modify. In addition, this shorthand only works on one instance of a string; it will not be propagated to every instance of the replaced string in the command line. Including a few extra characters to obtain a unique string guarantees that the substitution occurs at the place you intended.

For example, if you enter the erroneous command line

```
cs /usr/bin/new.file /usr/personal/new.file
```

the shell prints the message

```
Command not found.
```

At the next shell prompt, you can change your command line as follows:

```
^cs^cp^
```

This substitutes the correct command (cp) for the misspelled version, and executes the correct command line.

5.4 Editing and reexecuting previous commands

When you want to reexecute a previous command with a slightly different command line, you may invoke and edit a command line using the following notations:

!*{identifier}*x

Repeat the most recent command specified by *identifier* with *x* appended to it directly without intervening space. *identifier* may be the history number of a command or the string beginning a command. *x* may be a character or a string. For example,

```
more file1
```

may be reinvoked on file1A with the notation

```
!{m}A
```

This invokes the command

```
more file1A
```

The braces may be omitted if the string to be appended (*x*) begins with a space or if the resulting string unambiguously picks out a command from the history list. For example, if the current history list is as follows:

```
261 mail
262 vi note
263 mail fred < note
264 rm note
265 ls
266 cd manual/texts
267 ls -l
268 vi chap.1
269 make chap.1
270 more chap.1
271 history
```

In this situation the history substitution

```
!ma
```

will reinvoke command number 269, not the command

```
more chap.1a
```

as you might have expected. To invoke this latter command, you could have given

```
!{m}a
```

```
!n:s/x/y/
```

Repeat the *n*th command and substitute *y* for *x*. *x* and *y* may be characters or strings. This has the same effect as the up-arrow command discussed above, except it can be done on any previous command. For example,

```
cat file1 | lp
```

may be edited using the notation

```
!cat:s/file1/file2/
```

This invokes the command

```
cat file2 | lp
```

!n:gs/x/y/

Repeat the *n*th command and replace every instance of *x* with *y*. *x* and *y* may be characters or strings. Where *x* is a string, this is global substitution. For example,

```
nroff file1 > outfile1 &
```

may be edited using the notation

```
!!:gs/file1/file2/
```

This invokes the command

```
nroff file2 > outfile2 &
```

Where *x* is a character, only the first instance of *x* per word will be replaced by *y*. For example, if the command

```
echo 111 2211
```

is edited using the notation

```
!!:gs/1/3/
```

This invokes the command

```
echo 311 2231
```

When you use the **!** notation, a character or characters following a colon (as in **:s** or **:gs**) are called **modifiers**. They are used to modify previous command lines. See “Using Modifiers With Your Command History.” Another use of modifiers is described in “Variable Substitution.”

5.5 Reusing parts of previous command lines

The following history notations use special notations or numeric modifiers to refer to parts of a command line:

!\$ Refers to the last word on the preceding command line. For example, after the command

```
mv file1 /usr/bin
```

you may use the notation

```
cd !$
```

to invoke the command

```
cd /usr/bin
```

!n:x Refers to the *x*th argument of the *n*th command, where *n* is the (history) number of a previous command. For example, if the following command is number 5 in your history listing:

```
nroff file1 > outfile.1&
```

the first argument of the command line (the filename) may be referred to using the notation

```
wc -l !5:1
```

(where 5 is the history number). This invokes the command

```
wc -l file1
```

because *file1* is the first argument to the *nroff* command referenced by the number 5. (The command name *nroff* is the zero-th argument here.)

!n^ Refers to the first argument of the *n*th command, where *n* is the history number of a previous command. This is the equivalent of

```
!n:1
```

5.6 Using modifiers with your command history

The C shell provides modifiers that can be used to modify previous command lines. A modifier is a colon followed by one or more characters. The sections show how to use modifiers to substitute text or refer to parts of a previous command line. This section describes

modifiers that perform a variety of other functions, including changing arguments and affecting how the shell evaluates your new command.

The following are possible modifiers:

- `:h` Remove the last pathname component, leaving the head. See “Variable Substitution” for examples of how to use this modifier.
- `:t` Remove all leading pathname components, leaving the tail. See “Variable Substitution” for examples of how to use this modifier.
- `:r` Remove an extension filename component (`.xxx`), leaving the root name. See “Variable Substitution” for examples of how to use this modifier.
- `:e` Remove all but the extension filename component (`.xxx`). This modifier does not work in conjunction with the `history` command; see “Variable Substitution” for examples of how to use it.
- `:s/x/y/` Substitute the string `y` for `x`. See “Editing and Reexecuting Previous Commands.”
- `&` Repeat the most recent substitution.
- `:g` This modifier must be followed by one of the substitution modifiers (`s` or `&`). It indicates that the substitution will be applied globally. See “Editing and Reexecuting Previous Commands.”
- `:p` Print the command but do not execute it. For example,

```
!v:p
```

prints your most recent `vi` command but does not reexecute it. You can use the `:p` modifier to determine the effect of editing a command; for example, to change a previous `vi` command to an `ls` command and print the command instead of executing it:

```
!vi:s/vi/ls/:p
```

The shell prints

```
ls prog.c
```

This becomes your “most recent command” and you may execute it using the notation

```
!!
```

- :q Quote substituted words and prevent further substitution. See “Variable Substitution” for examples of how to use this modifier.
- :x Quote substituted words but allow blank interpretation. See “Variable Substitution” for examples of how to use this modifier.

These modifiers can be combined with each other, as with the :gs and :g& global modifiers, or with the :p no-execute modifier. The :h, :t, :r, and :e modifiers may also be used in combination with one another. For instance, if command number 15 in the history list is

```
cat /etc/termcap
```

Then

cd !15:1:h	expands to	cd /etc
cat !15:1:t	expands to	cat termcap
ls !15:1:h:t	expands to	ls etc

5.7 Other uses for command history

You can use the history mechanism (!) to set your C shell prompt so it will increment sequentially at each command, beginning at one. See the explanation of the `prompt` variable under “C Shell Variables” for details.

You can also use the command

```
repeat n command
```

to repeat *command* *n* times. *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list (see “Using Shell Metacharacters” for an explanation of these terms). Input and output redirection occurs once, even if *n* is 0. For example,

to execute the `date` command three times, you can use the command

```
repeat 3 date
```

If you use a large number by mistake and the command starts repeating many times, you can send an *interrupt* to stop the process.

6. Using shell metacharacters

Shell **metacharacters** are characters that perform special functions in the shell. This section discusses how to use these metacharacters. The following are the C shell metacharacters:

- ! An exclamation mark invokes the history mechanism. See “Listing and Reusing Commands.”
- ~ A tilde is used as the first part of a directory name. It is replaced with either your home directory (if it is used alone or followed by a pathname below your home directory such as `~/project/phase1`) or the home directory of another user (if it is followed by the login name of that user, such as `~lori`). See “Specifying Home Directories” for details.
- & An ampersand at the end of a command line causes the shell to run the command(s) in the background and prints the process ID(s).
- ? A question mark used as part of a file or directory name causes the shell to match any single character (except a leading period).
- * An asterisk used as part of a file or directory name causes the shell to match zero or more characters (except a leading period).
- [] Brackets around a sequence of characters (except the period) cause the shell to match each character one at a time. The shell will not match a leading period, even if the period is included within the brackets.
- A hyphen used within brackets to designate a range of characters (for example, `[A-Z]`) causes the shell to match each character in the range.
- { } Braces around a series of filenames cause the shell to perform an action on each file in the series. The filenames must be separated

by commas.

- < A less-than sign following a command and preceding a filename causes the shell to take the command's input from that file.
- > A greater-than sign following a command and preceding a filename causes the shell to redirect the command's standard output into the file. When followed by an ampersand (>&), it causes the shell to redirect the command's standard error output to the same file as standard output. See "Input and Output" for a description of how to redirect standard output and standard error output using > and >&.
- >> Two greater-than signs following a command and preceding a filename cause the shell to append the command's output to the end of an existing file. When followed by an ampersand (>>&), they cause the shell to redirect the command's standard error output to the end of the same file as standard output. See "Input and Output Redirection" for a description of how to redirect standard output using >>.
- | A vertical bar between two commands on a command line causes the shell to redirect the output of the first command to the input of the second command. This can occur multiple times on a command line, forming a pipeline.
- ; A semicolon between two commands on a command line causes the shell to execute the commands sequentially in the order in which they appear.
- () Parentheses around a pipeline or sequence of pipelines cause the whole series to be treated as a simple command (which may in turn be a component of a pipeline), and a subshell to be spawned for the commands' execution.
- \ A backslash prevents the shell from interpreting the metacharacter that follows it.
- ' ' Single quotes around a command, a command name and argument, or an argument prevent the shell from interpreting the enclosed metacharacters.

- " " Double quotes around a command, a command name and argument, or an argument prevent the shell from interpreting the enclosed metacharacters. Parameter substitution and command substitution are still performed. See “Quoting.”
- ` ` Back quotes around a command cause the characters in that command to be replaced with the output (via standard output) from that command.

6.1 Specifying home directories

You can use the tilde (~) as the initial character in a filename or pathname to avoid typing the absolute or relative pathnames of home (login) directories. An initial tilde in a pathname, for example,

```
~/chapter2
```

indicates a file below your own home directory. When the command is executed, the tilde is replaced by the value of your environmental variable HOME. A tilde followed by the login name of another user, for example,

```
~virginia/chapter2
```

indicates the login name of that user and will be replaced by the absolute pathname of that user’s home directory.

You can use this notation when giving a pathname as an argument to any command; for example,

```
cp ~virginia/memol ~/memos/virginia.memo
```

6.2 Specifying filenames with metacharacters

Using the filename expansion metacharacters (also called “wildcards”) will spare you the job of typing long lists of filenames in commands, looking to see exactly how a filename is spelled, or specifying several filenames that differ only slightly.

These metacharacters are interpreted and take effect when the shell evaluates commands. At this point, the word incorporating the metacharacter(s) is replaced by an alphabetic list of filenames, if any are found that match the pattern given. Filename expansion metacharacters can be used in any type of command, except in the filenames given for input and output redirection. To turn off the special

meaning of metacharacters and use them as ordinary letters, they must be quoted. See “Quoting.”

The following are filename expansion metacharacters in the C shell:

- ? A question mark matches any single character in a filename. For example, if you have files named

```
a bb ccc dddd
```

the command

```
echo ???
```

matches a sequence of any three characters and returns

```
ccc
```

- * An asterisk matches any sequence of characters, including the empty sequence, in a filename. (It will not, however, match the leading period in such files as `.login`.) To list the sequence of files named

```
chap chap1 chap2 chap3 chap3A chap12
```

you can use the notation

```
ls chap*
```

The files are listed as

```
chap chap1 chap12 chap2 chap3 chap3A
```

Note that in the first file listed, `chap`, the asterisk matched the null sequence composed of no characters.

- [] Brackets enclosing a set of characters match any *single* character, one at a time, from the set of enclosed characters. Thus,

```
ls chap.[12]
```

matches the filenames

```
chap.1 chap.2
```

Note that this does not match `chap.12`. To match filenames

chap.10, chap.11, and chap.12, use the notation

```
chap.1[012]
```

You can also place a hyphen (-) between two characters in brackets to denote a range. For example,

```
ls chap.[1-5]
```

is the equivalent of

```
chap.[12345]
```

A range of characters can also be indicated in brackets. The notation [a-z] matches any lowercase character, [A-Z] matches any uppercase character, and [a-zA-Z] matches any character, regardless of case.

{ } Braces specify that the enclosed strings (separated by commas) are to be consecutively substituted into the containing characters. For example,

```
A{xxx,yyy,zzz}B
```

expands to

```
AxxxB AyyyB AzzzB
```

This expansion occurs before any other filename expansion, and the results of each expanded string are sorted separately, preserving left-to-right order. A typical use of this would be

```
mkdir ~/{work,home,consult}
```

to make the subdirectories `work`, `home`, and `consult` in your home directory. This notation may also be nested. For example, the following command provides a quick way to see what executable programs are located in the usual places on an A/UX system:

```
ls /{bin,usr/{bin,games}}
```

None of these metacharacters will match the initial period at the beginning of special files such as `.login`. These must be matched explicitly. Periods that do not begin a filename can be matched by metacharacters.

If you use these metacharacters and the shell fails to match an existing filename, it displays the message

```
No match.
```

6.3 I/O redirection

An executing command may expect to accept input and create output, possibly including error output (error messages). In the A/UX system, there are default locations set for input and output:

- Standard input is taken from the terminal keyboard.
- Standard output is printed on the terminal screen.
- Standard error output is printed on the terminal screen.

These defaults can be changed using the following metacharacters (also called **redirection symbols**).

- < A less-than sign followed by a filename “redirects standard input” (takes command input from a file or device other than the keyboard). (The name of the file has variable, command, and filename expansion performed on it first.) For example,

```
mail ellen < note
```

uses a file named `note` instead of a message typed from the keyboard as the input to `mail`.

- << *word*

Two less-than signs followed by a word make the shell read input up to a line that is identical to *word*. Filename expansion, variable substitution, and command substitution are not performed on *word*, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting mechanism (`\`, `"`, `'`, or ```) appears in *word*, variable and command substitution are performed on the intervening lines, allowing `\` to quote `$`, `\`, and ```. Commands that are substituted

have all blanks, tabs, and newlines preserved, except for the final newline, which is dropped. The resulting text is placed in an anonymous temporary file, which is given to the command as standard input.

- > A greater-than sign followed by a filename “redirects standard output” (prints command output in a file or to a device other than the terminal screen). If a file by that name does not exist, a new file is created; otherwise the file’s previous contents are overwritten. For example,

```
sort file1 > file2
```

uses a file for the output of the `sort` command. When `sort` completes, *file2* contains the sorted contents of *file1*. Several variants are also available. For the `>` symbol, if the variable `noclobber` is set, then the file must not exist or be a character special file (for example, a terminal or `/dev/null`), or an error results. This helps prevent accidental destruction of files. In this case, the `>!` form can be used to suppress this check. The form `>&` routes the diagnostic output into the specified file as well as the standard output. The form `>&!` both suppresses `noclobber` and routes the diagnostic (as well as the standard) output into the specified file. In all these forms, *name* is expanded in the same way as `<` input filenames are.

See “Input and Output” for more information on redirecting standard error output.

- >> Two greater-than signs followed by a filename append the output of a command to a file. If no file by that name exists, one is created. For example,

```
who >> log
```

appends the output of the `who` command to the end of the existing file `log`. Again, variants are available. If the variable `noclobber` is set, then it is an error for the file not to exist unless one of the `!` forms, either `>>!` (put at end of file and clobber) or `>>&!` (put, with diagnostics, at end of file and clobber) is given. The `>>&` form puts error (as well as standard)

output at the end of the named file. Otherwise, all these forms are similar to `>`.

6.4 Combining commands: pipelines

You can send the output of one command as input to another command by using the vertical bar (`|`). When two or more commands are joined by a vertical bar, the command line is called a **pipeline**.

For example, to see which files in a directory contain the sequence `old` in their names, you can use a pipeline as follows:

```
ls | grep old
```

The pipe character (`|`) tells the shell that output from the first command (the list of files produced by the `ls` command) should be used as input to the `grep` command. The output of the pipeline (filenames in the current directory containing the string `old`) prints on standard output (unless you redirect it to a file).

Pipelines may consist of more than two commands; for example,

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

prints the number of files in the current directory whose names contain the string `old`.

Pipelines may also be executed in the background. For example, to avoid the time-consuming process of waiting for a very large file to be sorted and printed, you could give the following pipeline:

```
sort mail.list | lp &
```

This pipeline would sort the contents of a file named `mail.list` and send the sorted information to the `lp` program to be placed on the printer queue. The shell would respond with the process ID of the last command in the pipeline.

The `tee` command is a “pipe fitting”; it can be put anywhere in a pipeline to copy the information passing through the pipeline to a file. See `tee(1)` in *A/UX Command Reference* for more information.

A **filter** is a program or a pipeline that transforms its input in some way, writing the result to the standard output. For example, the `grep` command finds those lines that contain some specified string and prints

them as output.

```
grep 'correction' draft1
```

prints only the lines in `draft1` that contain the string `correction`.

Filters are often used in pipelines to transform the output of some other command. For example,

```
who | grep jon
```

prints

```
jon          ttyt8          Jul 21 12:25
```

if a user whose login name is `jon` is currently logged into the system on `ttyt8`.

6.5 Command grouping

You can use the following metacharacters to group commands:

- ;** Group several commands on one command line by separating one command from another with a semicolon (`;`). The commands will be executed sequentially in the order in which they appear. For example, the command line

```
cd test; ls
```

changes to the `test` directory and then lists its contents.

- &** Group background commands on a single line by separating them with ampersands (`&`) and then ending the line with another ampersand. The background commands will exit independently while the shell continues to accept new commands in the foreground.
- ()** Enclose a group of commands in parentheses to execute them as a separate process in a subshell (a new instance of the shell). For example,

```
(cd test; rm junk)
```

first invokes a new instance of the shell. This shell changes the directory to `test` and then removes the file `junk`. After this, control is returned to the parent shell, where the current directory

has not changed. Thus, when execution of the commands is over, you are still in your original directory.

The commands

```
cd test; rm junk
```

(without the parentheses) are executed in the current shell and have the same effect but leave you in the directory `test`.

6.6 Conditional execution

You can use the following symbols to indicate that your command should be executed only if some condition is met:

`&&` The command form

```
command1 && command2
```

means “If *command1* executes successfully (returns a zero exit status), then execute *command2*.”

`||` The command form

```
command1 || command2
```

does the reverse. This form means “If *command1* does not execute successfully (returns a nonzero exit status), then execute *command2*.”

For exit status, see “Exit Status: The Value of the Command.”

Conditional execution is also available in joining pipelines. For other ways of obtaining conditional execution, see “Control-Flow Constructs.”

6.7 Quoting

If you need to use the literal meaning of one of the shell metacharacters or control the type of substitution allowed in a command, use one of the following **quoting mechanisms**:

`\` A backslash preceding a metacharacter prevents the shell from interpreting the metacharacter. For example, to use the `echo` command to display a question mark, you must precede the question mark with a single backslash (`\`). Thus,

```
echo \?
```

prints

```
?
```

Without the backslash, the `echo` command would generate a list of all one-character filenames in the current directory. If there are none, the command returns

```
?
```

- ' ' Single quotes prevent the shell from interpreting any metacharacters in the enclosed string. The command

```
echo '*test'
```

prints

```
*test
```

while the command

```
echo *test
```

attempts to list all the files in your current directory ending with the characters `test`. If there are none, the command returns

```
*test
```

- " " Within double quotes, variable substitution and command substitution occur, but filename expansion and the interpretation of blanks do not. For example, if you have the variable `message1` with the value `"this is a test"`, the command

```
echo "$message1"
```

prints

```
this is a test
```

Double quotes can also be used to give a multiword argument to commands; for example,

```
echo "type a character"
```

For information on variable substitution, see “Arguments and Shell Variables.” You can also suppress filename expansion universally by setting the `noglob` environment variable. See “C Shell Variables.”

- A command name enclosed in back quotes is replaced by the output from that command. This is called **command substitution**. For example, if the current directory is `/users/marylyn/bin`, the command

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

is equivalent to

```
set i=/users/marylyn/bin
```

If a back quote occurs within the command to be executed, you must escape it with a backslash (`\``); otherwise the usual quoting conventions apply within the command.

Command substitution takes place before the filenames are expanded. If the output of substituted command is likely to be more than one word, the command must be enclosed in double quotes as well as back quotes; for example,

```
set a="`head -1` /dev/tty"
```

where the `head` command might yield more than one word. The double quotes in this example preserve the blank spaces from the input.

7. Working with more than one shell

When you wish to use another A/UX shell, you can use one of the following commands:

`sh` This spawns an instance of the Bourne shell.

`ksh` This spawns an instance of the Korn shell.

`csh` This spawns another instance of the C shell.

You can type these at your shell prompt; for example,

```
csh
```

In this case, your new shell will run as a **subshell** or “child” of your current one. You can use the `exit` command or the `eof` sequence to return to your original login shell whenever you wish. (If you have the `ignoreeof` C shell variable set, you must use the `exit` command; the `eof` sequence will not work to exit the C shell. See “C Shell Variables.”)

7.1 Changing to a new shell

You can also obtain a new shell using the `exec` command; for example,

```
exec sh
```

If you use the `exec` command, the Bourne shell program `sh` replaces your current shell. You cannot return to your original shell; it has disappeared. You can, of course, use the command

```
exec csh
```

to get a new copy of the C shell.

Generating new instances of a shell affects the environment settings for each shell. See “The Environment and New Shell Instances” for more information.

7.2 Changing your default shell

To change your default shell from the C shell to the Bourne or Korn shell, use the `chsh` command. For example,

```
chsh login.name /bin/ksh
```

(where *login.name* is your login name on this system) changes your default login shell to the Korn shell. See `chsh(1)` in *A/UX Command Reference* for more information.

8. The environment

The **environment** is a list of variables and other data that is available to all programs (including subshells) invoked from the shell. A shell inherits the environment that was active when it started, and passes the

environment (including any modifications you make to the environment) to all programs it invokes.

You can modify the environment using the `setenv` command (see “Adding Environment Variables and Modifying Values.”)

Note: Modifying the environment in a subshell (for example, in a shell script) never changes the parent shells or their environments. Values in the environment are *copied* to subshells’ environment, and any changes there are made only to the copies.

The most essential environment variables are assigned default values during login or by the shell every time you invoke it. Convenient but inessential variables are simply left unassigned. Thus a default environment is created, which you can redefine by resetting the default values or adding new elements.

8.1 Global environment variables

The C shell maintains a list of environment variables that are required by the A/UX shells. In addition, any variable that you create or modify using the `setenv` command is part of the environment and is passed to new instances of the shells (see “Adding Environment Variables and Modifying Values”).

Note: Global environment variables in the C shell pass among instances of all three A/UX shells (the C shell, the Bourne shell, and the Korn shell).

8.1.1 Listing existing values

To print a list of your current environment, use the command

```
printenv
```

This prints a list such as

```
HOME=/users/doc/elaine
PATH=/bin:/usr/bin:
EXINIT=set wm=10
LOGNAME=elaine
SHELL=/bin/csh
MAIL=/usr/mail/elaine
TERM=mac2
```

8.1.2 Adding environment variables and modifying values

You may create new environment variables or modify the value of existing ones using the command

```
setenv name value
```

For example,

```
setenv j 22
```

creates an environment variable (*j*) with the value 22. This variable can be referenced and used in the current shell and its subshells.

Environment variables can be modified using `setenv` at the shell prompt or in your `.login` file (see “The `.login` File.”) For example, to modify your `PATH` variable to include more pathnames, use the command

```
setenv PATH /etc:/usr/bin:/bin:/usr/uch
```

8.1.3 Removing environment variables

You can remove environment variables in the C shell using the command

```
unsetenv name ...
```

8.1.4 Commonly used environment variables

The following variables are typically inserted into the environment. By convention, environment variable names are uppercase. Some of these variables are assigned default values at login or by the shell at invocation. All of them can be reset by the user.

HOME At login this variable is set to the pathname of your home directory. Its value is the default argument (home directory) for the `cd` command. `~` is another name for

\$HOME.

- PATH** The default value for **PATH** is the current directory, `/bin`, and `/usr/bin`. A valid **PATH** value is a list of directory names separated by colons. Whenever you give a command, the shell checks the directories specified by your **PATH** variable to locate the command and execute it. If the directory containing the command file is not specified, the shell will not locate the command. **PATH** is usually set in the `.login` file. For efficiency, the list of directories in the **PATH** variable should be in order from the directories containing commands most often used to those least often used. If you add a command to one of the directories in **PATH** other than the current directory, you must give the `rehash` command, or the shell will not be able to find the commands. See “A Sample `.login` File” for an example.
- EXINIT** The value of this variable can be set to various options for your editing environment when you are using the `ex` or `vi` text editing program. See “Using `ex`” and “Using `vi`” in *A/UX Text Editing Tools*, and “A Sample `.cshrc` File.”
- LOGNAME** This variable contains your login name.
- MAIL** The value of this variable is set to the pathname of the file where your mail is received. This variable is typically set in the file `.login` in the user’s home directory.
- SHELL** The value of this variable indicates the shell that is invoked when you log in (your **login shell**). It is set at login using the information found in the `/etc/passwd` system file. In A/UX, if no shell is specified in `etc/passwd`, the default shell is the Bourne shell. For instructions on how to change your login shell, see `chsh(1)` in *A/UX Command Reference*.
- TERM** This variable specifies the type of terminal you are using. For A/UX systems, the default is set to `mac2`. You can see the value of your **TERM** variable using the command

```
echo $TERM
```

8.2 C shell variables

The C shell also maintains variables that are only relevant to the C shell (and will be ignored by the other shells). If these variables are created or modified at the shell prompt, they are valid only for the current shell. However, if they are assigned a value in the `.cshrc` file, they will be available to all new instances of the C shell.

Note: Because the C shell reads your `.cshrc` file every time a new instance of the C shell is invoked without the `-f` flag option, variables that have been set in your `.cshrc` file will be available in new instances. Although they are not technically “in the environment,” they will still be in effect for every instance of the C shell with the value specified in the `.cshrc` file.

See “Shell Variables” for more information on using variables in the C shell.

8.2.1 Listing existing values

The command

```
set
```

lists the value of all your current C shell variables.

8.2.2 Adding C shell variables and modifying values

C shell variables are set using the command:

```
set name [=value]
```

For example,

```
set history=200
```

If you use the `set` command at the shell prompt to modify a value or create a new variable, your variable assignments remain local to the shell you are currently working in (see “Adding Environment Variables and Modifying Values”).

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable that is not set. The following substitutions may not be modified with `:` modifiers:

`$?0` Substitutes 1 if the current input filename is known, 0 if it is not.

`$<` Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

8.2.3 Removing C shell variables

Any C shell variable can be removed using the `unset` command:

```
unset name
```

8.2.4 C shell variables

The following variables are typically assigned a value in the `.cshrc` file. This makes them available to all instances of the C shell. Some of these variables are assigned default values at login or by the shell at invocation. All of them can be reset by the user.

<code>argv</code>	Set to the arguments given to the shell. It is from this variable that arguments are substituted; that is, <code>\$1</code> is replaced by <code>\$argv[1]</code> , and so forth.
<code>cdpath</code>	<pre>set cdpath=<i>path</i></pre> Lists alternate directories searched to find subdirectories in <code>chdir</code> commands.
<code>cwd</code>	<pre>set cwd=<i>path</i></pre> Lists the full pathname of the current directory. This variable is set by the shell to <code>cwd= `pwd `</code> .
<code>echo</code>	<pre>set echo</pre> Causes each command and its arguments to be printed on the screen just before execution. For user-defined commands, all expansions occur before printing. Built-in commands are printed before command and filename substitution, because these substitutions are then done selectively. Set when <code>csh</code> 's <code>-x</code> command line option is given.

histchars set histchars *string1 string2*
 Changes the characters used in history substitution: *string1* replaces ! and *string2* replaces ^ .

history set history=*n*
 The value of this variable is a number specifying how many previous command lines are saved. In the A/UX standard distribution, *history* is given an initial value of 200. If you assign a very large number to this variable (for example, 500), it will use up a lot of memory.

home set home=*dir*
 Contains the home directory of the invoker, initialized from the environment. The filename expansion of ~ refers to this variable.

ignoreeof set ignoreeof
 If set, the shell ignores an *eof* from the keyboard. This prevents shells from accidentally being killed by typing the *eof* character.

ignoreexit set ignoreexit
 If set, the shell ignores an *exit* from the keyboard.

mail set mail=[*n*] *mailfile* ...
mailfile is the file the shell checks for mail. By default, it checks for mail every ten minutes after producing a shell prompt. If *mailfile* has been modified since you last accessed it, it prints the message "You have new mail." Supplying a number (*n*) before *mailfile* specifies a new interval (in seconds) to wait before checking for mail. If you have more than one *mailfile*, your mail message reads "New mail in *mailfile*."

noclobber set noclobber
 Restrictions are placed on output redirection to ensure that files are not accidentally overwritten or destroyed, and that >> redirections refer to existing files.

noglob	<p>set noglob</p> <p>If set, filename expansion is inhibited. This is most useful in shell scripts that are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.</p>
nonomatch	<p>set nonomatch</p> <p>If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed.</p>
notify	<p>set notify</p> <p>Notifies you when your background job completes, without waiting until the next prompt.</p>
path	<p>set path=<i>path</i>...</p> <p>Each <i>path</i> specifies a directory to search to execute commands. A null <i>path</i> specifies the current directory. If you don't specify a path variable, only full pathnames execute. The default search path is <code>./bin, /usr/bin, and /usr/ucb</code>. The default path for the superuser is <code>/etc, /bin, /usr/bin, and /usr/ucb</code>. If you start the shell without the <code>-c</code> or <code>-t</code> flag option, it will hash the contents of the directories in the path variable after reading <code>.cshrc</code> and each time you reset the path variable. If you add new commands to these directories while the shell is active, you may have to give the <code>rehash</code> command before these commands are found.</p>
prompt	<p>set prompt=<i>string</i></p> <p>The value of this variable is <i>string</i>. This string is printed at the beginning of a line, indicating that the shell is ready to receive input. By default, this is set to <code>%</code>. If this variable is set to <code>\!</code> the prompt will be the history number of the current command line; this is usually set the in <code>.cshrc</code> file. If you do not define the <code>prompt</code> variable in your <code>.cshrc</code> file, it</p>

will be set by the `/etc/cshrc` file in the A/UX standard distribution. See “Using Your `.cshrc` File.”

- `savehist` `set savehist=n`
Saves *n* entries from the history list in the file `~/.history` when you log out. This is read into your history list when you next log in. If *n* is too large, it slows down the shell during startup.
- `shell` `set shell=file`
file contains the default shell to use for executing shell files. This is used in forking shells to interpret files that have execute bits set but are not executable by the system. Note that this will only affect scripts starting with the number sign (`#`), because others are passed to the Bourne shell.
- `status` `set status=n`
Returns the status of the last command. 0 indicates success of a built-in command, 1 indicates failure of a built-in command, and 0200 is added to the status of a command terminated abnormally. Note that this variable is almost never set explicitly. The `exit(2)` system call sets it, as does the `exit` built-in command.
- `time` `set time=n`
Prints the execution statistics at the completion of any command running over *n* CPU seconds. These statistics include user, system, and real times, and the ratio of user plus system times to real time. There is also a corresponding `time` command that can be used to time a given command or shell.
- `verbose` `csh -v`
Causes the words of each command to be printed after history substitution. Set when the `csh` command line option is given.

8.3 The environment and new shell instances

Because the C shell reads the `.cshrc` file each time it starts up, the values you have defined there are available to the new C shell. Any values you have assigned using the `setenv` command will also be available in a new instance of the C shell (invoked without the `-f` flag option).

If you have assigned values to variables using the `set` command at the shell prompt (or within a shell script), these remain local to the shell in which you assigned them. Because these changes are made to a *copy* of the parent shell's environment, the parent shell's environment is never affected by changes in a subshell, even if you use the `setenv` command. Note, however, that changes made using `setenv` in a shell will be passed on to subsequent new instances of the shell. When a subshell terminates, its environment no longer exists.

Note that the `.login` file is read only once, at login. Thus, if you have changed the value of an environment variable, the subshell will inherit the new value, not the value set routinely in `.login`. You can force a new instance of the shell to read `.login` by using the `source` command; see “Executing Shell Scripts.”

8.4 Special environments

Normally, the environment for a command is the complete environment of the shell where the command was given. You can change the environment used by a command with the A/UX command

```
env [-][name=value ...][command][args]
```

to set the environment for *command*. With this command, you can not only add things to the environment inherited by a command, but also exclude the current environment. To add variables and their values to the current environment, give the variables and values before the command name. For example, to run a subshell with a changed `PATH` environment variable, you could give the command

```
env PATH=directory-list sh
```

For the duration of the new shell (and its subshells), the `PATH` variable would be set to the directories in the list.

To set up a completely new environment, first give the option `-`, which excludes the current environment, and then assign the variables and values you want. These (and only these) will be available in the environment for the new command.

8.5 The default environment on your system

Whenever you log in, the following procedures occur:

- The login program sets the variables `HOME` and `SHELL` from the information in the system file `/etc/passwd`.
- The login program then checks the `/etc/cshrc` file to find out the default environment to set up for all users.
- The **login shell** (the shell that is automatically invoked when you log in) assigns default values (for example, to `prompt` and `history`).

When you invoke new instances of the shell (for example, using the `csh` command), the new shell checks the environment for any new values you may have placed there for these variables. If it doesn't find any values in the environment, it assigns the default values.

Then the new shell reads your `.cshrc` and `.login` files. If you have assigned new values there, it uses your values instead of the defaults.

- The C shell reads the `.cshrc` file every time it starts up, not only at login. Use the `.cshrc` file to set C shell variables and to define aliases you wish to be available across all invocations of the C shell. All variables and aliases set in this file are available to new instances of the C shell as if their values were in the environment. However, none of the local values set in this file are available to instances of the Bourne shell or Korn shell.
- The C shell reads the `.login` file when you log in. This file usually contains values for environment variables that should be available to all instances of the shell, including Bourne shell and Korn shell.

9. The `.login` file

The `.login` file is simply a text file. It contains a series of commands typed exactly as you would type them at the shell prompt. When you

log in, the C shell looks in your home directory for files named `.cshrc` (see “The `.cshrc` File”) and `.login`. When the shell finds one or both of these files, it executes all the commands found there before issuing the shell prompt. If no `.login` or `.cshrc` file exists, your environment will simply be the default environment created by the shell at login.

9.1 A sample `.login` file

The following is a sample `.login` file:

```
setenv PATH :/bin:/usr/bin:/etc:/usr/new:~/bin
setenv EXINIT "set wm=10"
date
ls
```

The variables and commands in this file are discussed in the sections that follow.

9.1.1 Locating commands

The `PATH` environment variable lists the directories where the shell will look for the executable files that are A/UX (or user-defined) commands. Each time you give a command, the shell searches the directories listed in the order specified. Most A/UX commands are located in the `/bin` or `/usr/bin` directory. When you assign a value to `PATH`, be sure to include these directories.

If the shell cannot find the file in one of the directories specified, the command cannot be executed and you will see the message

```
Command not found.
```

If you do not know the directory containing a particular A/UX command, see `whereis(1)` in *A/UX Command Reference*.

A valid `PATH` value is a list of directory names (specified by absolute pathnames), separated by colons. If the list of directories begins with a colon, the path search begins in the current directory. At login, the `PATH` variable is set as follows:

```
setenv PATH :/bin:/usr/bin:/usr/ucb
```

This assignment sets the `PATH` variable to the current directory and the system directories `/bin`, `/usr/bin`, and `/usr/ucb`.

To reset the `PATH` variable in the `.login` file, insert a line such as

```
setenv PATH :/bin:/usr/bin:/usr/ucb:/etc:/usr/new:~/bin
```

The `setenv` command is discussed under “Adding Environment Variables and Modifying Values.”

If you include the pathnames of personal directories that contain shell programs you have written, these will be accessible to the shell no matter what your current directory is. If you wish to execute a command or shell program that is not in one of the directories in your `PATH` variable, simply give the absolute pathname of the directory where the command or shell program is to be found.

For information on referencing variables using the `$` syntax (as in `$HOME` above), see “Variable Substitution.” For more information about pathnames, see the glossary in *A/UX System Overview*.

9.1.2 Your editing environment

The `EXINIT` environment variable tells the shell how to initialize the `vi` or `ex` editing programs. It is set to a series of editor commands that should be run every time the editor starts up. In the sample `.login` above, for example, the command

```
setenv EXINIT "set wm=10"
```

assigns the value of `EXINIT` as the command

```
set wm=10
```

This command sets the word-wrap margin so that the editor will automatically break lines ten spaces before the right margin. The command is enclosed in double quotes because the entire string must be treated by the C shell as one “word” and not divided up.

For details on `EXINIT`, see *A/UX Text Editing Tools*. For the use of double quotes, see “Quoting.”

9.2 Customizing your login procedure

You can also use your `.login` file to customize your login procedure. In the sample `.login` above, the commands


```
date
ls
```

direct the shell to display the date and time and then list all the files in the current directory before displaying the shell prompt. These will be executed at login.

You can include any commands you wish in `.login`, including your own shell scripts.

10. The `.cshrc` file

The `.cshrc` file is similar to the `.login` file, but is normally read at every invocation of the C shell. Thus, any definitions you include in this file will be available to every instance of the C shell.

10.1 A sample `.cshrc` file

The following is a sample `.cshrc` file:

```
set prompt='\!: '
set ignoreeof
alias lc ls -C
```

These commands are described below.

10.1.1 Using history numbers as your prompt

The C shell history mechanism keeps track of your command lines by a number, automatically incrementing the number each time you give a command. If you use this number as your prompt, it is more convenient to refer to previous commands by number (see “Listing and Reusing Commands”).

In your `.cshrc` file, the command

```
set prompt='\!: '
```

sets your C shell prompt to the history character `!` followed by a colon and a blank space. This will print as a shell prompt a number that increments with each command:

```
1: ...
2: ...
```

Note: The `!` must be escaped (preceded by a backslash) and enclosed in single quotes to keep the shell from interpreting it at the wrong time, for example, when it reads and executes your `.cshrc` file.

10.1.2 Protection against unintentional logout

The shell terminates, logging you out of the system, when it recognizes the *eof* sequence. This can cause you to log out inadvertently when sending mail or using any other program that also terminates when you type an *eof*.

To prevent this, you may set the `ignoreeof` variable in your `.cshrc` file. This causes the shell to ignore *eof* from the keyboard.

When this variable is set, you must use the `logout` or `exit` command to log out.

11. Aliases for commonly used commands

The C shell `alias` command renames existing commands or creates a name for a long command line. Aliases can be defined at the shell prompt or in the `.cshrc` file.

The C shell keeps a list of aliases. Each time you give a command, the first word of the command is compared with the list. If it is an alias name, then it is replaced with the definition of that alias. You can use an alias to redefine any shell or A/UX command except `alias`; however, it is not advisable to redefine keywords such as `foreach` or `while`.

11.1 Defining an alias

You define an alias with the command

```
alias name definition
```

where *name* may begin with any printable character, but the rest of the characters must be letters, digits, or underscores (generally it is a good idea to avoid using `/`, `:`, `*`, `?`, and so on), and *definition* may contain any valid commands, including shell scripts and metacharacters. Note that *definition* cannot include another alias. If *definition* includes spaces, the whole command must be enclosed in quotes.

For example, the alias

```
alias lc 'ls -C'
```

causes the `lc` command to produce output as if you had typed

```
ls -C
```

which displays its output in columns. The alias definition is quoted because it contains a blank.

Note that the invention of a new command name, `lc`, allows you to use both `ls` (in any form desired) *and* `lc`.

Alias definitions can also include all shell metacharacters, variables, arguments, command substitution, and so forth.

For example,

```
alias prtsort 'sort *.list'
```

creates a command `prtsort`. When you type

```
prtsort
```

the command line

```
sort *.list
```

executes, sorting files in the current directory that end in the characters `“.list”`.

The use of double quotes in an alias definition allows certain expansions to occur at the time the alias is defined. For example, the definition

```
alias lshome "ls $HOME"
```

allows for the variable expansion of `$HOME`. Then if you type

```
lshome
```

you see a listing of your home directory.

When you create aliases at the shell prompt, they are not exported to the environment. To make aliases available to every invocation of the C shell or any script run with separate shell, put their definitions in the

.cshrc file, which is normally read every time a C shell starts up.

11.2 Listing and removing aliases

The `alias` command with no arguments lists all aliases that have been defined in your environment.

Aliases can be removed with the command

```
unalias name(s)
```

11.3 Aliases that take arguments

It is also possible to define aliases that accept arguments and contain multiple commands or pipelines. The following alias definition instructs the shell to invoke an `ls` command after any `cd` (change directory) command. This alias will accept an argument (a directory name or pathname) where `\!*` occurs in the alias.

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

The history notation for accepting an argument is explained as follows:

- `\!` The history character (!) is preceded by a backslash (\) to prevent its default meaning when the command is invoked.
- `\!*` The (`\!*`) here indicates that an argument will be substituted at this place in the command and that it is not considered an error if no argument is given.

The `alias` command uses history substitution and modifiers in a variety of ways. Because the `cd` command will function without an argument (changing to the user's login directory), the correct notation in our example is "`\!*`". If you use either "`\!:1`" or "`\!^`", the alias will require an argument in order to execute without an error message. For example,

```
% alias j 'echo my favorite pastime is \!:1'  
% j walking  
my favorite pastime is walking
```

However, it is an error if you omit the argument using the `\!:1` notation:

```
% j
bad ! arg selector
```

If you use the `\!*` notation, the argument is optional. If you supply an argument to the alias, it works as you would expect:

```
% alias j 'echo my favorite pastime is \!*'
% j walking
my favorite pastime is walking
```

It is not an error if you omit the argument in this case:

```
% j
my favorite pastime is
```

12. Shell execution options

The shell is a program like other A/UX commands, and it too has a variety of options used to control how it executes. All shell execution options can be specified on the command line when you invoke a new shell or run a shell script with the `cs`h command

```
cs h -opt[opt...] [file]
```

This invokes a subshell or runs a script (*file*) with the options specified.

The C shell execution options are as follows:

- c Commands are read from the (single) following argument *file*, which must be present. Any remaining arguments are placed in `argv`. This cannot be nested.
- e The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.
- f The shell will start faster, because it doesn't search for or execute commands from the `.cshrc` file in your home directory. Some scripts may fail if executed using the `-f` option because of aliases and variables that will not be read from `.cshrc`.
- i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Without this option, a shell is interactive if its standard input and standard output are a

terminal.

- n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.
- s Command input is taken from the standard input stream.
- t A single line of input is read and executed. A \ may be used to escape the newline at the end of this line and continue onto another line.
- v The `verbose` variable is set, with the effect that command input is printed after history substitution.
- x The `echo` variable is set, so that commands are printed immediately before execution.
- V The `verbose` variable is set even before `.cshrc` is executed.
- X The `echo` variable is set even before `.cshrc` is executed.

If arguments remain after the execution options are processed (but you did not specify the `-c`, `-i`, `-s`, or `-t` option), the first argument is taken as the name of a file containing commands to be executed. The shell opens this file and saves its name for possible resubstitution by `$0`. Remaining arguments initialize the variable `argv`.

13. Job control

C shell job control allows you to suspend current jobs, move a foreground job to the background (and vice versa), check on the status of background jobs, refer to specific background jobs by number, change a job's status, and receive notification when a job is done.

Every job you run in the C shell is associated with a **job number**; for example, when you give a background command, such as

```
diff file1 file2 >> file3 &
```

the job number (in brackets) displays before the process ID:

```
[3] 12345
```

Job numbers are assigned sequentially, so the first job is 1, the second job is 2, and so forth.

13.1 Suspending a job

To suspend your current foreground job, type the *suspend* character. See “Canceling Commands” for the A/UX standard distribution *suspend* character. When you type the *suspend* character, it sends an immediate stop signal to the current job; pending output and unread input are discarded.

When the shell interprets *suspend*, it prints a message in the form

```
[job-number] + Stopped name
```

where *job-number* is the job number of the current job; + indicates that it is the current job; and *name* is the command name of the stopped job. For example,

```
[2] + Stopped diff
```

13.2 Listing jobs

You can list your jobs with the command

```
jobs
```

Your jobs will be listed and their status (running or stopped) will be indicated like this:

```
[3] + Running lp chapter1 &  
[2] - Stopped vi chapter2  
[1]  Running diff file1 file2 > diff.file &
```

The + indicates the current job, and the - indicates the preceding job.

If you include the -l option, process IDs will be shown as well as the job numbers:

```
jobs -l
```

13.3 Changing the status of stopped jobs

Once you have a stopped job, you can give another command at the shell prompt (leaving the job suspended), resume the job in the foreground, resume another stopped job, or continue the command processing in the background.

To leave a job suspended, do nothing. When you give the command

jobs

you will see it listed as `Stopped`. To run a stopped job in the background, give the command

```
bg %job-number
```

For example,

```
bg %2
```

The `bg` command with no argument

```
bg
```

puts the most recent stopped job in the background to continue executing. If a job number is given as an argument to `bg`, it must be preceded by a percent sign (%). The following notation is available for job numbers:

<code>%job-number</code>	refers to a specific job by number
<code>%+</code>	refers to the current job
<code>%-</code>	refers to the preceding job
<code>%string</code>	refers to the most recent stopped job that began with those characters

As a shorthand notation, just naming a job, with an ampersand, resumes that job in the background. In addition, `%*&` resumes all stopped jobs in the background.

Thus, if the most recently stopped job was an `lp` command whose job number was 4, you could resume this job in the background with any of the following commands:

```
bg
bg %+
bg %4
bg %lp
%4&
```

After one of these commands, you would be shown the command line of the job that was being put in the background, and then the shell

prompt would be returned.

A job running in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to send output to the terminal, but this can be disabled by giving the command

```
stty tostop
```

This causes background jobs to stop when they try to send output, just as they do when they try to read input.

If a background job needs neither input nor output and completes execution in the background, the shell displays a message in the form

```
[job-number] + Done name
```

For example,

```
[2] + Done diff
```

You can bring a job to the foreground with the command

```
fg %job-number
```

The same conventions for referring to a stopped job given above under the `bg` command work for the `fg` command. The `fg` command works exactly like `bg`. Just naming a job brings it into the foreground, so saying `%1` brings job 1 to the foreground. Similarly, saying `%*` brings all stopped jobs to the foreground. Once your job is in the foreground, you can continue working as before.

13.4 Blocked jobs

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible. If the shell is busy with another process when it learns about a blocked job, it will wait until it is about to print another prompt before displaying a message.

13.5 Canceling jobs

To cancel a job, use the command

```
kill [%]number
```

number can be either a process ID, or a job number preceded by a percent sign (%). The rules about job numbers that apply to `bg` and `fg`

also apply to the `kill` command. Using the `kill` command with process IDs to cancel jobs is discussed in “Canceling Background Commands.” Thus if you had a current background `lp` job whose job number was 4, you could cancel this job with any of the following commands:

```
kill %+
kill %4
kill %lp
```

The shell will display a message that the job has been terminated:

```
[4] + Terminated lp bigfile &
```

13.6 Logging out with stopped jobs

If you try to log out while your jobs are stopped, you will be warned with

```
You have stopped jobs.
```

If you use the `jobs` command to see what the stopped jobs are, or if you immediately try to log out again, the shell will not warn you a second time. The stopped jobs will be terminated.

14. Using shell layering

Many C shell users will not wish to use shell layering, since job control performs essentially the same functions while maintaining your environment. However, if you do wish to use shell layering with the C shell, you should make sure the `switch` and `susp` characters are defined to different control sequences. Otherwise job control will function correctly in the shell layer you invoke, but the `sh1` program will be inaccessible. The A/UX standard distribution sets `switch` to `CONTROL-^` and `susp` to `CONTROL-z`. To check that these are defined to different control sequences on your system, enter the command

```
stty -a
```

at the shell prompt. This displays the settings for various user-definable sequences. See `stty(1)` in *A/UX Command Reference* for additional details.

See Chapter 5, “Shell Layering,” for more information.

15. Overview of shell programming

A shell program is simply a list of commands. These commands can be entered at the prompt or inserted in a file. They may contain

- variables and assignments
- control-flow statements (for example, `if`, `for`, `case`, or `while`)
- built-in shell commands
- any A/UX command

Input for the shell program may be read from the keyboard (this is the default standard input), taken from files, or embedded in the program itself (see “Taking Input From Scripts”).

Shell programs may write output to the terminal screen (the default standard output), to files, or to other processes (via pipes).

When the shell program executes, each command is executed until the shell encounters either an *eof* character or a command delimiter that directs it to stop. During execution, you can trap errors and take appropriate action.

15.1 Writing shell programs

You can enter a shell program at the prompt. When you use a built-in shell command that expects a delimiter (such as `end`) or a certain type of input, a question mark appears after you press RETURN, on each line until you give the expected delimiter; for example,

```
% foreach i ([A-Z]*)
? cat $i
? end
%
```

Note that you can send an *interrupt* to cancel the script and return to the primary prompt.

You can also write a shell program in a text file (using a text editor), and then execute it (see “Executing Shell Scripts”). These program

files are often called **shell scripts**. Note that all shell programs may be entered at the shell prompt or inserted in a file. This does not affect their actions. Hereafter “shell scripts” will be used to refer to shell programs that reside in a file.

15.2 Executing shell scripts

There are several ways to execute a shell script; these differ mostly in terms of whether or not a new instance of the shell is invoked.

- You can use the `csh` command to read and execute commands contained in a file. The script will be run in a “subshell,” which means that it will have access to only the values set in the environment and will be unable to alter the parent shell. The command

```
csh filename args ...
```

causes the shell to run the script contained in *filename*. Shell scripts run with the `csh` command can be invoked with all the options possible for the `set` command.

- You can change the mode of the shell script file to make it executable. For example,

```
chmod +x filename
```

makes *filename* executable. Then the command

```
filename args ...
```

has the same effect as using the `csh` command. The script is run in a Bourne subshell, which means that it will have to use Bourne shell syntax. See Chapter 2, “Bourne Shell Reference.” If your script uses C shell built-in commands, it will not execute successfully in the Bourne shell. If the first line of your shell script is a `#` and your current shell is the C shell, the script will be run in the C shell.

- You can run a shell script inside the current shell by using the `source` command. The `source` command tells the current shell to run the script; no subshell is invoked. This should be used if you wish to use local shell variables or functions, or modify the current shell:

`source filename args ...`

Because the commands are executed in the current shell, this is the way to run a script that is to change values in the shell.

- You can run executable shell scripts or A/UX commands with the `exec` command. This should be used when the shell script program is an application designed to execute in place of the shell and replace interaction with it:

`exec filename args ...`

In this case, the script or command *replaces the current shell*. This means that when the script is over, control will not return to the shell. If you were in a login shell, you will be logged out.

15.3 Comments

A word beginning with a number sign (#) causes that word and all the following characters up to a newline to be ignored.

15.4 Writing interactive shell scripts

A shell script can invoke an interactive program such as the `vi` editor. If standard input is attached to the terminal, `vi` will read commands from the terminal and execute them just as if invoked from an interactive shell. After the session with `vi` is finished, control will pass to the next line in the script. In a similar manner, a script can invoke another copy of a shell (using `sh`, `csh`, or `ksh`), which will interpret commands from the terminal until it receives an *eof*. Control will be returned to the script. You can use this to create a special environment for certain tasks by setting environment variables in a shell script and then invoking a new subshell.

You can also write interactive shell scripts by using the `line` and `echo` commands. See “Reading Input” and “Writing to the Standard Output.”

15.5 Canceling a shell script

You can cancel a shell script just like an ordinary A/UX command. If the script is running in the background, use the `kill` command. See “Canceling Commands” for details on `kill` and various types of interrupts that can stop a command.

Note: Interrupts can be handled within the script using the `onintr` command. See “Fault Handling and Interrupts.”

15.6 Writing efficient shell scripts

In general, built-in commands execute more efficiently than A/UX commands. See “Summary of C Shell Commands” at the end of this chapter for a complete list of these commands. The following built-in commands are useful in constructing efficient shell scripts:

`rehash` This causes the shell to remember the search path of any new commands.

`times` This prints the accumulated user and system times for processes.

You can also set the `-f` shell execution option using

```
csh -f script
```

This will prevent the new shell instance from reading `.cshrc`. You should only use this if your script does not require any of the settings in `.cshrc`.

Careful setting (or resetting inside a shell script) of the `PATH` environment variable make sure that the most frequently used directories are listed first. This also improves efficiency.

16. Command evaluation

When you give a command, the shell evaluates the command in one pass and then executes it. To force more than one pass of evaluation, use the `eval` command described in “Summary of C Shell Commands.”

While evaluating the command, the shell performs the following substitutions on variables:

- History substitution. This checks every word of the command for a word beginning with `!` and replaces that word with the elements of history specified. For more information, see “Listing and Reusing Commands.”

- **Alias substitution.** This checks the first word of every command to see if it is an alias (a user-defined name for another command). If an alias is found, it is replaced by the text of the alias. Only one check for aliases is made, so that an alias itself cannot contain an alias. For information on aliases, see “Aliases for Commonly Used Commands.”

- **Tilde substitution.** This replaces an initial tilde with a directory name (see “Specifying Home Directories”). The following forms are recognized:

~ This is replaced by the value of the HOME variable.

~*name* This is replaced by the home directory of another user (where *name* is the user’s login name).

- **Variable substitution.** This replaces variables preceded with \$ (for example, \$USER) with their values. Only one pass of evaluation is made. For example, if the value of the variable d is daphne, then the command

```
echo $d
prints
daphne
```

However, if the value of the variable d is \$name, then the command

```
echo $d
prints
$name
```

The second variable is never evaluated and the value is not substituted. See “Variable Substitution” for more information.

- **Command substitution.** The shell replaces a command enclosed in back quotes with the command’s output. For example, if the current directory is /users/doc/virginia, then the command

```
echo `pwd`  
prints  
/users/doc/virginia
```

- Blank interpretation. The shell breaks the characters of the command line into words separated by blank spaces or tabs. The null string is not regarded as a word unless it is quoted; for example,

```
echo ''
```

passes the null string as the first argument to `echo`, whereas the commands

```
echo
```

and

```
echo $null
```

(where the variable `null` is not set or set to the null string) pass no arguments to the `echo` command.

- Filename expansion. The shell scans each word for filename expansion metacharacters (see “Using Shell Metacharacters”) and creates an alphabetical list of filenames that are matched by the pattern(s). Each filename in the list is a separate argument. Patterns that match no files are left unchanged.

These evaluations also occur in the list of words associated with a `foreach` loop.

16.1 Command execution

After all substitution has been carried out, commands are executed as follows:

- Built-in commands and shell scripts run with the `source` command are executed in the current shell. The command has available all current shell execution options, the values of variables and environment variables, and functions defined in the current shell.

- A/UX commands, programs, executable shell scripts, shell scripts run with the `csH` command, and series of commands enclosed in parentheses are executed in a subshell. The current shell invokes a child shell that executes the commands and then returns control to the parent shell. Only the values in your environment are available to these processes.
- Commands and executable scripts run with the `exec` command execute in place of the current shell.

If the A/UX command or program name does not specify a pathname, the environment variable `PATH` is used to determine what directories should be searched for the command. The only exception to this is built-in commands.

For more information about the execution of shell scripts, see “Executing Shell Scripts.”

16.2 Exit status: the value of the command

Although there are exceptions, in general a command’s exit value is zero (0) if it executes successfully, and its exit value is nonzero if it terminates abnormally. In some cases, a command exits with a nonzero exit status with a normal termination; for example, the `diff` command returns nonzero exit status if it finds no differences between two versions of a file. The shell saves the exit value of the commands in the variable `status`. The exit status is used primarily in shell scripts as `$status`. See `signal(3)`, `exit(2)`, and `wait(2)` in *A/UX Programmer’s Reference* for the values of various exit statuses.

17. Arguments and shell variables

A shell script may use two types of variables:

Arguments: Arguments given on the command line are stored as elements in the special variable `argv`, and as the parameters `$1, ... $n`.

Shell variables: Shell variables may be simple strings or arrays of strings. These variables can be assigned on the command line or inside the script.

The relationship between variables inside a shell script and existing shell variables depends on how the script is run. See “Executing Shell Scripts.” In all cases, shell scripts have access to the variables and values in the environment.

17.1 Arguments

The shell stores the arguments you give to a script sequentially as elements of the one-dimensional array `argv`.

When you enter any command at the prompt, the shell stores the elements of the command line as follows: the command name is stored in `argv[0]`, the first argument is stored in `argv[1]`, the second argument in `argv[2]`, and so forth. Thus, for the command

```
diff letter1 letter2
```

`argv[1]` is the word `letter1` and `argv[2]` is the word `letter2`. For the command

```
echo "not a directory"
```

the phrase

```
not a directory
```

is assigned to `argv[1]`, whereas the command

```
echo not a directory
```

assigns each word to a position in `argv`.

This means that the arguments (for example, filenames) used in the script can be given on the command line when the script is run. For example, the command line

```
script arg1 arg2
```

assigns `argv[0]` to `script`, `argv[1]` to `arg1`, and `argv[2]` to `arg2`. These may also be referenced as `$0`, `$1`, and `$2`, respectively. To refer to all `argv` values, you may use `$*`, which is equivalent to `argv[*]`.

17.2 Shell variables

The C shell supports only string variables. Variables can be simple strings or arrays of strings. They can be assigned values anywhere in

the script. Variable names begin with a letter and consist of letters, digits, and underscores.

17.2.1 Assigning values

You can assign values to variables using the `set` command with the syntax

```
set name=value
```

Blanks and/or tabs may surround the equal sign. All values are stored as strings. Command substitution and filename expansion will be performed on *value*. It is an error to attempt to use a variable that has not been set.

To set a variable to a string of words separated by blanks, the entire string must be enclosed in double quotes; for example,

```
set longvar="this is a long variable"
```

The double quotes prevent the shell from carrying out blank interpretation and breaking up the phrase to be assigned into its constituent words. Without the quotes, the phrase would be considered five words and could not all be assigned to one variable.

After the variable assignments

```
set user="fred stone" set box='???' set acct=18999
```

the following values are assigned:

```
user   = fred stone
box    = ???
acct   = 18999
```

Because the C shell supports only string variables, all of these values (including 18999) will be strings of characters. Note that the question mark metacharacters must be quoted with single quotes to prevent pattern matching.

A variable may be set to the null string with the syntax

```
set name
```

Arrays are initially set with the command

```
set name = (word...)
```

The array is created and its elements are set to the words inside the parentheses. The first element of the array is assigned *word1*, the second element is assigned *word2*, and so forth. Subscripting of elements begins with 1. The words must be separated by spaces. They are treated like the values assigned simple string variables. If a word itself is to contain spaces, it must be quoted.

Existing individual elements of arrays already assigned values with the `set` command can be assigned new values with the command

```
set array[subscript] = value
```

The array element whose *subscript* is given is assigned *value*. Subscripts begin with 1. *value* is treated just like the values assigned to simple string variables.

Shell variables can be set and used interactively to provide abbreviations for frequently used strings. For example, the sequence of commands

```
set b=/usr/fred/bin
mv file $b
```

moves `file` from the current directory to the directory `/usr/fred/bin`.

17.2.2 Changing position of elements

The command

```
shift [name]
```

renumbers the elements of the array whose name is given. Elements 2, 3, 4... are renumbered as 1, 2, 3..., and so forth. The first element is discarded. This can be useful, for example, when working through a list of files. After each file is processed, a shift is performed and the next filename becomes argument 1.

If *name* is not given, `shift` operates on `argv`.

17.2.3 Removing shell variables

Remove variables using the `unset` command followed by the name of the variable:

```
unset name
```

The variable and its value will both be removed.

17.3 Variable substitution

Variables, arrays, and the special variable `argv` are referenced and their values are substituted when the identifier (the variable name or array or `argv` element) is preceded by a dollar sign (`$`):

\$identifier

Here *identifier* is one of the following:

variable-name

This can be the name of any simple string variable; for example,

```
$j1 $1 $8 $version
```

This will substitute the value of the variable. For example, after the command

```
set form=last
```

the command

```
echo $form
```

prints

```
last
```

array-name

This can be the name of any array; the entire array will be substituted. For example, after setting up the array `address` with

```
set address=(333 Delaney St)
```

the command

```
echo $address
```

prints

```
333 Delaney St
```

subscripted-array-element

Subscripted names of an array or `argv` elements in the form

name [*subscript*]

print the value of that element. *subscript* can be another variable, a number, or a range of numbers separated by `-`, where the first number, if omitted, will be assumed to be 1 and the second number will be assumed to be the last element. For example,

```
$argv[1] $names[1-3] $argv[-12]
$names[1-] names[$choice]
```

A special shell variable, `*`, can be used to substitute for all elements of arrays or `argv`. Note that this differs from the usual “filename expansion” usage of the asterisk character (`*`).

number

A `$` followed simply by a digit will be taken as referring to that element of `argv`. For example,

```
$1
```

refers to the first element of `argv`.

- * A `$` followed simply by `*` will be taken as referring to all elements of `argv`.

\$#name

This substitutes the number of elements (words) in the variable whose name is given.

The form

```
$(identifier)
```

is equivalent to `$(identifier)` and can be used with all of the above forms. It is used when the *identifier* is followed by a letter or digit. For example,

```
set tmp=/tmp/ps
ps a >${tmp}a
```

substitutes the value of the variable `tmp` and directs the output of `ps` to

the file /tmp/psa, whereas

```
ps a >$tmpa
```

causes the value of the variable `tmpa` to be substituted.

For all forms of substitution, you can use the following modifiers. The modifiers are shown below in examples that assume the following variable substitution:

```
set i=/usr/mail/marylyn
echo $i
/usr/mail/marylyn
```

:h Remove trailing pathname, leaving only the head.

```
% echo $i:h
/usr/mail
```

:t Remove leading pathname, leaving only the tail.

```
% echo $i:t
marylyn
```

:e Remove root filename, leaving only the extension.

```
% set a=oem.address
% echo $a:e
address
```

:r Remove filename extension, leaving only the root.

```
% echo $a:r
oem
```

:q Quote substituted words, prevent further substitution.

```
% set a='t*'
% ls $a
t.1 t.2 t.3 t.4
% ls $a:q
t* not found.
```

:x Quote substituted words, but allow blank interpretation.

```

% set a='echo *'
% $a
chap.1 chap.2 t.1 t.2 t.3 t.4
% $a:q
echo *: command not found.
% $a:x
*
```

The modifiers `:h`, `:t`, and `:r` can be prefixed with `g` (`:gh`) for global modification. If braces are used, the modifiers must be inside. Only one modifier is allowed for each substitution. Substitutions of environment variables may not include modifiers.

17.4 Testing assignment

If a variable is not set, an error will be reported. For example, if the variable `d` is not set,

```
echo $d
```

or

```
echo ${d}
```

prints

```
d: Undefined variable
```

The following structures allow you to test whether variables are set and not null.

```

$?name
${?name}
```

For both of these, the value `1` is substituted if *name* is set; and `0` is substituted if *name* is not set.

17.5 Variables set by the system

The following variables are set by the C shell during execution:

`status`

The exit status of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise they return a nonzero exit status. This is used in the `if` and `while` constructs for control of execution.

\$ The process ID of this shell in decimal. Because process IDs are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example,

```
ps a > /tmp/ps$$  
commands  
rm /tmp/ps$$
```

18. Control-flow constructs

The shell has a variety of ways of controlling the flow of execution. The actions of the `foreach` loop and the `switch` branch are determined by data available to the shell. The actions of the `while` loop and “if then else” branch are determined by the exit status returned by commands or tests. Control-flow constructs can be used together and loops can be nested.

In the following constructs, reserved words such as `end` are only recognized following a newline or semicolon. *command-list* is a sequence of one or more simple commands separated or terminated by a newline or a semicolon.

18.1 foreach loops

To repeat the same set of commands for several files or arguments, use the `foreach` loop:

```
foreach name (word ...)  
    command-list  
end
```

For each iteration of the loop, *name* is set to the next *word* and then *command-list* is executed. If no *word* is given, the elements of `argv` are used.

To terminate a loop before the end of *word*, or to continue a loop and cause it to reiterate before the end of *command-list* is reached, use the loop-control commands

```
break  
continue
```

These commands can appear only between the loop delimiters. The `break` command terminates execution of the current loop; execution

resumes after the nearest subsequent end. The continue command causes execution to resume at the beginning of the current loop.

18.2 switch statements

A multiway conditional branch is provided by the switch command, whose form is

```
switch (word)
  case pattern:
    command-list
  breaksw
  ...
  case pattern:
    command-list
  breaksw
  default pattern:
    command-list
  breaksw
endsw
```

word is matched against each *pattern*. If a match is found, *command-list* after that pattern is executed. Otherwise *command-list* after default (if provided) is executed.

Each *command-list* must end with breaksw; this breaks out of the case statement after execution.

Patterns may include filename expansion metacharacters. To be used literally, pattern-matching metacharacters must be quoted.

18.3 while loops

The while command allows a loop that depends on whether or not a certain condition is met.

A while loop has the form

```
while expression
  command-list1
end
command-list2
```

The value tested by the `while` command is the exit status of *expression*. Each time *expression* returns a status of zero (true), *command-list1* is executed. The loop terminates when *expression* returns a nonzero exit status; then *command-list2* is executed.

To terminate a loop otherwise, or to proceed to the next loop test before the end of *command-list1* is reached, use the loop-control commands

```
break
continue
```

These commands can appear only between the loop delimiters. The `break` command terminates execution of the current loop; execution resumes after the nearest subsequent end. The `continue` command causes execution to resume at the beginning of the current loop.

18.4 if then else

A general conditional branch is also available in the C shell, with the forms

```
if expression command
```

(The *command* in this form is a simple command.)

```
if expression then
  command-list1
[else if expression then
  command-list2]
[else
  command-list3]
endif]
```

The `if` command tests *expression* to see if it is not negative. If it is not (that is, if it is true), the commands following the `if` are executed; otherwise the commands following the `else` (if present) are executed.

Conditional execution of commands can also be achieved with the symbols `&&` and `||`. See “Conditional Execution” for details.

18.5 goto

The command

```
goto label
```

causes the shell to continue execution after the line consisting of *label*, which has the form

```
word:
```

label must be the only text on the line. It can be preceded by spaces or tabs.

18.6 exit

Shell scripts normally end when an *eof* is encountered. The exit status is that of the last command executed. The command

```
exit [expression]
```

can be used to cause termination. Exit status is set to *expression*. If *expression* is omitted, the exit status is that of the last command executed before `exit` was encountered.

19. Input and output

All forms of input and output redirection are allowed in shell scripts. If input or output redirection (using `<` or `>`) is done in any of the control-flow commands, the entire command is executed in a subshell. This means that any values assigned during execution of the command will not be available after the command is over and control returns to the parent shell. To avoid any problems this may cause, you can change standard input and output before the command begins with the `exec` command.

19.1 Standard error and output files

If you want to direct the error output of a command to a file (to save the error messages), use the syntax

```
ls filenames >& output
```

This writes both standard output and error output in the file `output`. If you want to save the command output and error output in separate files, use the syntax

```
(ls filenames > output) >& errors
```

19.2 Reading Input

The C shell does not have a built-in function for reading data from standard input; however, the `line` program can be used to provide this capability. Used in conjunction with the C shell `set` command, data from standard input can be stored in C shell variables. In the following example, the C shell variable “`a`” will contain the string “`hello, world`” after executing `line` to read data from standard input:

```
% set a = `line`  
hello, world  
% echo $a  
hello, world
```

See `line(1)` in *A/UX Command Reference* for more information.

19.3 Taking input from scripts

Input to a shell script can be embedded inside the script itself. This is called a **here document**. The information in a here document is enclosed as follows:

```
<<[-] word  
    information  
word
```

The first *word* may appear anywhere on a line; the second must appear alone and first on a line. The *words* must be identical and should not be anything that will appear in *information*. The second *word* is the *eof* for the here document.

Variable and command substitution will occur on *information*. Normal quoting conventions apply, so that `$` can be escaped with `\`. To prevent all substitution, quote any character of the first instance of *word*. (If substitution is not required, this is more efficient.) The choice of double or single quotes will be reflected in the resulting action.

To strip leading tabs and blanks from *word* and *information*, precede the first instance of *word* with the optional hyphen (`-`), as follows:

```
<<-word
```

Note: If you intend to indent your code, you must use the hyphen preceding *word* unless the commands you use can tolerate leading tabs and blanks.

For example, a shell procedure could contain the lines

```
foreach i
  grep $i /usr/lib/telnet
end
```

Here the `grep` command looks for the pattern specified by `$i` (in this case, the elements of `argv`) in the file `/usr/lib/telnet`. This file could contain the lines

```
fred mh0123
bert mh0789
```

An alternative to using an external file would be to include this data within the shell procedure itself as a here document:

```
foreach i
  grep $i <<!
  ...
  fred mh0123
  bert mh0789
  ...
!
end
```

In this example, the shell takes the lines between `<<!` and `!` as the standard input for `grep`. The second `!` represents the *eof*. The choice of `!` is arbitrary. Any string can be used to open and close a here document, provided that the string is quoted if white space is present and the string does not appear in the text of the here document.

Here documents are often used to provide the text for commands to be given for interactive processes, such as an editor, called in the middle of a script. For example, suppose you have a script named `change` that changes a product name in every file in a directory to a new name, as follows:

```

foreach i (*)
echo $i
ed $i <<!
g/oldproduct/s//newproduct/g
w
!
end

```

(Note that `ed` commands will not tolerate leading tab characters and there is no hyphen preceding the first *word*, therefore the code is not indented.) The metacharacter `*` is expanded to match all filenames in the current directory, so the `foreach` loop executes once for each file. For each file, the `ed` editor is invoked. The editor commands are given in the here document between `<<!` and `!`. They direct the editor to search globally for the string `oldproduct` and each time it is found substitute the string `newproduct`. After the substitution is made, the editor saves the new copy of the file with the `w` command.

You could make the change script more general by using parameter substitution as follows:

```

foreach i (*)
echo $i
ed $i <<!
g/$1/s//$2/g
w
!
end

```

Now the old and new product names (or any other strings) can be given as arguments on the command line:

```
change string1 string2
```

Substitution of individual characters can be prevented by using a backslash (`\`) to quote the special character `$`, as in

```

foreach i
echo $i
ed $i <<!
1, \${s}/$1/$2/g
w
!
end

```

This version of the script is equivalent to the first, except that the substitution is directed to take place on the first to the last lines of the file (1, \$) instead of “globally” (g) as in the first example. This way of giving the command has the advantage that the editor will print a question mark (?) if there are no occurrences of the string \$1.

Substitution can be prevented entirely by quoting the first instance of the terminating string; for example,

```
ed $i <<\'!
```

Note that backslash, single quotes and double quotes all have the same effect in this context: they turn off variable substitution and filename expansion.

To use leading tabs, precede the first *word* with a hyphen, as follows:

```

foreach i
    echo $i
    ed $i <<-!
        1, \${s}/$1/$2/g
    w
    !
end

```

19.4 Using command substitution

Command substitution can occur in all contexts where variable substitution occurs. You can use command substitution in a shell script to avoid typing long lists of filenames. For example,

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

runs the `ex` editor, supplying as arguments those files whose names end in `.c` and that contain the string `TRACE`.

Another example,

```
foreach i (`ls -t`)
    command-list
end
```

sets the variable *i* to each consecutive filename in the current directory, starting with the file that was most recently created or modified. The commands specified in *command-list* are then performed once for each file.

19.5 Writing to the standard output

The `echo` command is used to write to standard output (by default, the screen). The form of the `echo` command is

```
echo [-n] argument ...
```

The arguments are written to the standard output. They are evaluated like the arguments of any other command with variable and command substitution, filename expansion, and blank interpretation. Normal quoting conventions apply. Strings containing tabs or multiple blanks must be enclosed in double quotes. The arguments will be written sequentially separated by blanks, and unless the `-n` flag option is specified, they will be terminated with a newline.

If there are no arguments or the arguments are null variables, no output other than a blank line will ensue. If the arguments are unset, an error message will be printed.

If the `-n` flag option is specified, the output is written without a final newline.

20. Other features

20.1 Arithmetic evaluation

The C shell command `@` is used for integer arithmetic and to set variables to arithmetic expressions. The form of the `@` command is

```
@ variable = expression
```

variable can be a simple variable name or the subscripted element of an array. The possible *expressions* are listed in the next section. Each element in an expression must be surrounded by spaces. A simple

example of the @ command would be to increment a counter as follows:

```
@ i = $i + 1
```

20.2 Expressions

The C shell has operators similar to C, with the same precedence. These expressions are used in the @, exit, if, and while commands. The following operators are available in increasing precedence:

	logical (bit-wise) OR
&&	logical (bit-wise) AND
	binary OR
^	binary exclusive OR
&	binary AND
== != =~ !~	equal, not equal, equal, not equal
<= >= < >	comparison
<< >>	left shift, right shift
+ -	addition, subtraction
* / %	multiplication, division, modulus
!	logical negation
~	binary inversion or binary NOT

Note that many of these do not work with the @ construct.

Parentheses can be used to change operator precedence. The ==, !=, =~, and !~ operators compare their arguments as strings; all others operate on numbers. The operators =~ and !~ are like != and == except that the right-hand operand is a pattern (containing, for example, *s, ?s, and instances of brackets ([]) against which the left-hand operand is matched. This reduces the need for use of the switch statement in shell scripts when all that is really needed is pattern

matching.

Strings that begin with 0 are considered octal numbers. Null or missing arguments are considered 0. The result of all expressions are strings, which represent decimal numbers. Elements of expressions should be separated by spaces. The operators &, &&, |, ||, <, >, (, and) should be quoted to avoid interpretation by the shell.

Also available in expressions as primitive operands are commands enclosed in braces. (Note that the command must be surrounded by white space, for example “{ ls }”.) Commands execute successfully, returning true (that is, 1) if the command exits with status zero; otherwise they fail, returning false (that is, 0). If more detailed status information is required, then the command should be executed outside an expression and the variable `status` examined.

20.3 File status

The C shell allows inquiries about the status of files of the form

option name

Possible options are

- r read access
- w write access
- x execute access
- e existence
- o ownership
- z zero size
- f plain file
- d directory

Command and filename expansion are performed on the specified name and then it is tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all inquiries return false (0). For example, the form

`-e employees`

will return a true value (1) if the file `employees` exists; otherwise it will return 0.

21. Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively.

Execution of a command may fail for any of the following reasons:

- I/O redirection may fail if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a bus error or memory fault signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell will go on to execute the next command. An interactive shell will return to read another command from the terminal. If a shell script is being executed, the next command in the script will be read. Except for the last case, the shell will print an error message.

All other types of errors cause the shell to exit from a shell script. Such errors include

- Syntax errors, for example, “`if then done`”.
- A signal such as *interrupt*. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands.

The shell flag `-e` causes the shell to terminate if an error is detected.

21.1 Fault handling and interrupts

You can catch interrupts given to a shell script with the command

`onintr label`

When an interrupt is detected, execution will be transferred to the command following the line consisting of *label*, which has the form

word:

label must be the only text on the line. It can be preceded by spaces or tabs.

For example, `onintr` can be useful if you wish to clean up temporary files created by a shell script. After *label*, the commands to remove the temporary files and execute an `exit` command should be invoked.

21.2 Debugging a shell script

Several shell options can be set that will help with debugging shell scripts. These are

- e e (error) causes the shell to exit immediately if any command exits with a nonzero exit status. (This can be dangerous in scripts involving constructs where nonzero exit status is desired.)
- n n (no execute) prevents execution of subsequent commands. Commands will be evaluated but not executed. (Note that typing `csH -n` at a terminal will render the terminal useless until an *eof* is entered.)
- u u (unset) causes the shell to treat unset variables as an error condition.
- v v (verbose) causes the shell to print lines of a procedure as it reads them. Use this to help isolate syntax errors.
- x x (execution) provides an execution trace. Following variable substitution, each command is printed as it is executed.

The execution options can be turned on with the `csH` command if the script is executed as follows:

```
/bin/csh -option script
```

22. Summary of C shell commands

Input/output redirection is permitted for these commands. File descriptor 1 is the default output location.

`alias` [*name*][*word-list*]

Print aliases. With no arguments, this prints all aliases. The second form prints the alias for *name*. The final form assigns the specified *word-list* as the alias of *name*; *word-list* is command and filename substituted. *name* is not allowed to be `alias` or `unalias`. See “Aliases for Commonly Used Commands.”

`bg` [%*job*...]

Put the current or specified jobs in the background, continuing them if they were stopped. See “Changing the Status of Stopped Jobs.”

`break`

Cause execution to resume after the end of the nearest enclosing `foreach` or `while`. The remaining commands on the current line are executed. Multilevel breaks are thus possible by writing them all on one line. See “foreach Loops” and “while Loops.”

`breaksw`

Cause a break from a `switch`, resuming after the `endsw`. See “switch Statements.”

`case` [*label*:]

A label in a `switch` statement, as discussed below. See “switch Statements.”

`cd` [*name*]

If no argument is given, change to the home directory of the user. In the second form, change the shell’s working directory to directory *name*. If *name* is not found as a subdirectory of the current directory (and does not begin with `/`, `./`, or `../`), each component of the variable `cdpath` (see “List of C Shell Variables”) is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with `/`, this is tried to see if it is a directory.

`chdir` [*name*]

Another form of the `cd` command.

`continue`
Continue execution of the nearest enclosing `while` or `foreach`. The rest of the commands on the current line are executed. See “foreach Loops” and “while Loops.”

`default:`
Label the default case in a `switch` statement. The default should come after all case labels. See “switch Statements.”

`dirs`
Print the directory stack. The top of the stack is at the left, and the first directory in the stack is the current directory.

`echo [-n][word-list]`
Write the specified words to the shell’s standard output, separated by spaces and terminated with a newline unless the `-n` option is specified. See “Writing to the Standard Output.”

`else`
See the description of the `if` statement.

`end`
See the description of the `foreach` and `while` statements.

`endif`
See the description of the `if` statement.

`endsw`
See the description of the `switch` statement.

`eval [arg...]`
args are read as input to the shell and the resulting command(s) execute in the context of the current shell. This is usually used to execute commands generated by command or variable substitution, because parsing occurs before these substitutions. See “Command Evaluation.”

`exec [command]`
Execute the specified command in place of the current shell. See “Executing Shell Scripts” and “Changing to a New Shell.”

`exit [expr]`
Cause the shell to exit either with the value of the `status`

variable (first form) or with the value of the specified *expr* (second form). See `exit` under “Control-Flow Constructs;” also see “Protection Against Unintentional Logout” and “Working With More Than One Shell.”

`fg [%job...]`

Bring the current of specified jobs into the foreground, continuing them if they were stopped. See “Changing the Status of Stopped Jobs.”

`foreach name [(word-list)]`

...

`end`

Set the variable *name* successively to each member of *word-list* and execute the sequence of commands between this command and the matching `end`. (Both `foreach` and `end` must appear alone on separate lines.)

When the `foreach` command is read from the terminal, the loop is read up once, prompting with ? before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal, you can interrupt it. The built-in command `continue` may be used to jump to the next cycle of the loop, ignoring any subsequent commands in the loop. The built-in command `break` may be used to leave the loop immediately, discarding any remaining members of *word-list*. See “foreach Loops.”

`glob [word-list]`

Similar to the `echo -n` command (see “Writing to the Standard Output”), but no `\` escapes are recognized and words are delimited by null characters in the output. Useful for programs that use the shell to filename expand a list of words.

`goto [word]`

The specified *word* is filename and command expanded to yield a string of the form *label*. The shell rewinds its input as much as possible and searches for a line of the form *label* :, possibly preceded by blanks or tabs. Execution continues after the specified line. See `goto` under “Control-Flow Constructs.”

hashstat

Print a statistics line indicating 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 [*n*] [-h] [-r]

Display the history event list. Specifying *n* prints only the *n* most recent events. The `-h` flag option prints the history list without leading numbers. This produces files suitable for sourcing using the `-h` flag option to `source`. The `-r` flag option reverses the order of the printout to most recent first rather than oldest first. See "Listing and Reusing Commands," "Listing Previous Commands," and `history` under "C Shell Variables."

if [*expr*][*command*]

If the specified expression evaluates true, the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the `if` command. *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when *command* is not executed. Note that *expr* may be enclosed in parentheses.

if [*expr*] then

...

else if [*expr2*] then

...

else

...

endif

If the specified *expr* is true, the commands to the first `else` are executed; else if *expr2* is true, the commands to the second `else` are executed; and so on. Any number of `else-if` pairs are possible; only one `endif` is needed. The `else` part is likewise optional. (The words `else` and `endif` must appear at the beginning of input lines; the `if` must appear at the beginning of its input line or after an `else`.) See "Control-Flow Constructs."

`jobs [-l]`

List the active jobs. The `-l` flag option also lists process IDs. See “Job Control,” “Logging Out With Stopped Jobs,” “Checking Command Status,” “Listing Jobs,” and “Changing the Status of Stopped Jobs.”

`kill [-sig] [%job] [pid] [-l]`

Send either the terminate signal or the specified signal to the specified jobs or processes. Signals are given either by number or by name (as in `signal(3)` in *A/UX Programmer's Reference*, stripped of the prefix “SIG”). `kill -l` lists the signal names. There is no default; typing `kill` does not send a signal to the current job. If the signal being sent is terminate or hang up, the job or process is sent a continue signal as well. See “Canceling Commands,” “Job Control,” “Canceling Background Commands,” and “Summary of C Shell Commands.”

`login [name]`

Terminate a login shell, replacing it with an instance of `/bin/login`. This is one way to log out, included for compatibility with `sh(1)`.

`logout`

Terminate a login shell. Especially useful if `ignoreeof` is set. See “Protection Against Unintentional Logout.”

`nice[[+][-]number][command]`

Without an argument, lower the run priority for this shell to 4. The form

`nice +number`

or

`nice -number`

sets `nice` to the given number. The forms

`nice command`

and

`nice +number command`

run *command* at priority 4 and priority *number*, respectively. The superuser may increase a command's run priority by using

`nice -number command`

command is always executed in a subshell, and the restrictions place on commands in simple `if` statements apply. See `nice(1)` in *A/UX Command Reference* for more information.

`nohup [command]`

Without an argument, cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes running in the background with `&` are effectively run `nohup`. See "Logging Out."

`notify [%job]`

Notify you when the current or specified job completes without waiting for a prompt. The `notify` variable sets this automatically. See "List of C Shell Variables."

`onintr[-][label]`

Control the action of the shell on interrupts. Without an argument, `onintr` restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the terminal command input level. The form

`onintr -`

causes all interrupts to be ignored. The form

`onintr label`

causes the shell to execute a

`goto label`

when an interrupt is received or a child process terminates because it was interrupted (see the `label` command in this summary and "Fault Handling and Interrupts" for a description of the valid form of *label*) In any case, if the shell is running detached and interrupts are being ignored, all forms of `onintr`

have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

`popd [+n]`

Pop the directory stack, returning to the new top directory. With an argument *+n*, `popd` discards the *n*th entry in the stack. The elements for the directory stack are numbered from 0 starting at the top.

`pushd [name][+n]`

With no arguments, exchange 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 (as in `pwd`) onto the directory stack. With a numeric argument, rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

`rehash`

Cause the internal hash table of the contents of the directories in the `path` variable to be recomputed. This is needed if new commands are added to directories in the `path` while you are logged in. This should only be necessary if you add commands to one of your own directories, or if someone changes the contents of one of the system directories. See “Writing Efficient Shell Scripts.”

`repeat [count command]`

Execute the specified *command*, which is subject to the same restrictions as *command* in the one-line `if` statement above, *count* times. I/O redirections occur exactly once, even if *count* is 0. See “Other Uses for Command History.”

`set [name[index]=word]`

Without an argument, show the value of all shell variables. Variables that have a value other than a single word print as a word list in parentheses. The form

```
set name
```

sets *name* to the null string. The form

`set name=word`

sets *name* to the single *word*. The form

`set name [index]=word`

sets the *index*th component of *name* to *word*; this component must already exist. The form

`set name=word-list`

sets *name* to the list of words in *word-list*. In all cases the value is command and filename expanded. These arguments can be repeated to set multiple values in a single set command. Note, however, that variable expansion happens for all arguments before any setting occurs. See “C Shell Variables.”

`setenv name value`

Set the value of environment variable *name* to be *value*, a single string. The variable `PATH` is automatically imported to and exported from the `cs`h variable `path`; there is no need to use `setenv` for this. See “Adding Environment Variables and Modifying Values.”

`shift [variable]`

Shift the members of `argv` to the left, discarding `argv[1]`. It is an error for `argv` not to be set or to have less than one word as a value. The second form performs the same function on the specified variable. See “Changing Position of Elements.”

`source [-h[name]]`

Read commands from *name*. `source` commands may be nested; if they are nested too deeply, the shell may run out of file descriptors. An error in a `source` at any level terminates all nested `source` commands. Input during `source` commands is never placed on the history list. Normally input during `source` commands is not placed on the history list; the `-h` flag option causes the commands to be placed in the history list without being executed. See “Command Execution,” “Executing Shell Scripts,” and “The Environment and New Shell Instances.”

`stop [%job]`
Stop the current or specified job that is executing in the background.

`suspend`
Cause the shell to stop in its tracks, much as if it had been sent a *suspend* signal. This is most often used to stop shells started by `su` (see `su(1)` in *A/UX Command Reference*). You cannot suspend your login shell.

```
switch ([string])
case str1:
...
breaksw
...
default:
...
breaksw
endsw
```

Match each case label successively with the specified *string*, which is first command and filename expanded. The file metacharacters `*`, `?`, and `[...]` may be used in the case labels, which are variable expanded. If none of the labels match before a default label is found, the execution begins after the default label. Each case label and the default label must appear at the beginning of a line, and *string* must be enclosed in parentheses. The command `breaksw` causes execution to continue after the `endsw`. Otherwise control may fall through case labels and default labels as in the C programming language. If no label matches and there is no default, execution continues after the `endsw`. See “switch Statements.”

`time [command]`
With no argument, print a summary of time used by this shell and its children. If arguments are given, the specified simple command is timed and a time summary as described under the `time` variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

`umask [value]`
Display the file creation mask (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002, giving all access to the group and read and execute access to others, and 022, giving all access except no-write access to users in the group or others.

`unalias [pattern]`
Discard all aliases whose names match the specified pattern. Thus all aliases are removed by

```
unalias *
```

See “Listing and Removing Aliases.”

`unhash`
Disable use of the internal hash table to speed location of executed programs.

`unset [pattern]`
Remove all variables whose names match *pattern*. Thus all variables are removed by

```
unset *
```

See “Removing C Shell Variables.”

`unsetenv [pattern]`
Remove all variables whose name matches *pattern* from the environment. See also `setenv` above and `printenv(1)` in *A/UX Command Reference*. See “Removing Environment Variables.”

`wait`
Wait for all background jobs. If the shell is interactive, an *interrupt* can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

`while [(expr)]`
...
`end`
While the specified expression evaluates nonzero, evaluate the

commands between the `while` and the matching `end`. `break` and `continue` may be used to terminate or continue the loop prematurely. (The `while` and `end` must appear alone on their input lines.) Prompting occurs here the first time through the loop, as for the `foreach` statement, if the input is a terminal. See “while Loops.”

`%job-number[&]`

Bring the specified job into the foreground. Followed by an ampersand, continues the specified job into the background. See “Job Control.”

`@ [name[index]=expr]`

Without an argument, print the values of all the shell variables. The form

`@ name=expr`

sets the specified *name* to the value of *expr*. If the expression contains `<`, `>`, `&`, or `|`, at least this part of the expression must be placed within parentheses. The form

`@ name [index]=word`

assigns the value of *word* to the *index*th argument of *name*. Both *name* and its *index*th component must already exist.

The operators `*=`, `+=`, and so on are available as in the C programming language. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* that would otherwise be single words.

Special postfix `++` and `--` operators increment and decrement *name*, respectively. For instance, one way to increment a variable *i* is

`@ i++`

Chapter 5

Shell Layering

Contents

1. Invoking the sh1 program	1
2. Creating a shell layer	2
3. Suspending and resuming shell layers	2
4. Learning the status of shell layers	3
5. Deleting shell layers	3
6. Summary of sh1 commands	4

Chapter 5

Shell Layering

The `sh1` program allows you to create up to seven labeled subshells called **shell layers** within your login shell. These layers can then be referred to by name (or number), suspended and resumed, deleted, and so on. Each of these layers appears like your login shell, but can be used to run a process while you switch to another layer. This provides a management scheme for multiple concurrent processes.

When you are using the `sh1` program, you can suspend a shell layer (and the process you are running in that layer) by sending a *switch* character. This returns you to the `sh1` prompt where you can list other shell layers, resume a layer, delete a layer, and so on.

Note: If you are using the Korn shell or the C shell, you should make sure the *switch* and *susp* characters are defined to different control sequences. Otherwise, job control will function correctly in the shell layer you invoke, but the `sh1` program will be inaccessible. The A/UX standard distribution sets *switch* to CONTROL-` and *susp* to CONTROL-z. To check that these are defined to different control sequences on your system, enter the command

```
stty
```

at the shell prompt. This displays the settings for various user-definable sequences. See `stty(1)` in *A/UX Command Reference* for additional details.

1. Invoking the `sh1` program

To invoke the shell-layering facility, use the command

```
sh1
```

You will then see the `sh1` prompt:

```
>>>
```

2. Creating a shell layer

At the `sh1` prompt, you can create a new shell with the `create` command. Like all `sh1` commands, this can be abbreviated to the first letter of the command:

```
c [name]
```

This creates a new shell, where *name* may be a sequence of characters delimited by a blank, tab, or newline; only the first eight are significant. If you don't specify a name, the system will assign the number 1 for the first shell, 2 for the second, and so on, up to 7. Because the digits 1 through 7 are used for system-assigned names, they cannot be used for user-assigned names.

It is a good idea to name shells after the process you intend to run. For example, you can create a shell

```
c vi
```

in which you intend to use `vi`, and another shell

```
c machine.name
```

for a continuing `rlogin` session with another machine.

3. Suspending and resuming shell layers

The new shell layer uses the name you assigned it as a shell prompt. If you did not specify a name, it uses the number assigned by the system. When you see this prompt, you can begin working just as in your regular login shell.

To temporarily stop working in that shell, enter the *switch* sequence at the beginning of a line. (If you enter a *switch* in the middle of a line, the remainder of the information on that line will be discarded.)

You may use *switch* at the shell layer's prompt, or in the middle of an interactive job such as `vi`. Whatever you are doing in that layer will immediately be suspended, and the `sh1` prompt will be returned:

```
>>>
```

To continue working in a layer that you have stopped with *switch*, use the command

```
r name
```

For example,

```
r vi
```

brings your *vi* job back into the foreground. The shell layer resumes at the point where you suspended it. If you were in *vi*, it resumes *vi* at the same point in the file. However, you may need to use the *vi* CONTROL-L command to redraw your screen.

Note: When resuming a shell layer, you will not see a new prompt until you enter a second RETURN. If you give the resume command without an argument, the last layer you were working in will be resumed.

4. Learning the status of shell layers

You can obtain a listing of the current layers and their status by using the command

```
l
```

This returns output that looks something like

```
vi (02445) executing or awaiting input
```

where the number is a process ID. Used with the *-l* option, this command produces a listing similar to the *ps* command.

5. Deleting shell layers

When you delete a shell layer, all processes running in that layer are killed. If you are finished using a particular shell layer, you can remove it by leaving that layer using the *exit* command or *eof* instead of *switch*. Or you can remove a shell layer from the *sh1* prompt by using the *delete* command:

```
d name
```

6. Summary of sh1 commands

The following are the commands you can enter in response to the sh1 prompt. You can use either the full command name or just the first letter.

c[reate] *[name]*

Create a layer called *name* and make it the current layer. If you don't specify *name*, a layer will be created and assigned a digit between 1 and 7.

b[lock] *name [name ...]*

For each *name*, block the output of the corresponding layer when it is not the current layer.

d[ele]t[e] *name [name ...]*

For each *name*, delete the corresponding layer. All processes in the process group of the layer are killed (sent the hangup signal).

h[elp] or ?

Print the syntax of the sh1 commands.

l[ayers] [-l] *[name ...]*

For each *name*, list the layer name and its process group. The -l option produces a listing similar to the ps command. If no arguments are given, information is presented for all existing layers.

r[esume] *[name]*

Make the layer referenced by *name* the current layer. If no argument is given, the last existing current layer will be resumed.

t[oggle]

Resume the layer that was current before the last current layer.

u[nblock] *name [name ...]*

For each *name*, do not block the output of the corresponding layer when it is not the current layer.

q[uit]

Exit the `shl` program and return to the original login shell. All layers are killed (sent the hangup signal). After you exit the `shl` program, you will once again see the shell prompt.

name

Make the layer referenced by *name* the current layer.

Appendix A

Additional Reading

UNIX Shell Programming
Stephen G. Kochan and Patrick H. Wood
Hayden Books, 1985

10270455