CHAPTER 5

The Bourne Shell

MOTIVATION

The Bourne shell, written by Stephen Bourne, was the first popular UNIX shell and is available on all UNIX systems. It supports a fairly versatile programming language and is a subset of the more powerful Korn shell that is described in Chapter 6. Knowledge of the Bourne shell will therefore allow you to understand the operation of many scripts that have already been written for UNIX, as well as preparing you for the more advanced Korn shell.

PREREQUISITES

You should have already read Chapter 4 and experimented with some of the core shell facilities.

OBJECTIVES

In this chapter, I'll explain and demonstrate the Bourne-specific facilities, including the use of environment and local variables, the built-in programming language, and advanced I/O redirection.

PRESENTATION

The information is presented in the form of several sample UNIX sessions.

UTILITIES

The chapter introduces the following utilities, listed in alphabetical order:

expr   test

SHELL COMMANDS

The following shell commands, listed in alphabetical order, are introduced:

break	for..in..do..done	set
case..in..esac	if..then..elif..fi	trap
continue	read	while..do..done
export	readonly	

INTRODUCTION

The Bourne shell supports all of the core shell facilities described in Chapter 4, plus the following other facilities:

- several ways to set and access variables
- a built-in programming language that supports conditional branching, looping, and interrupt handling
- extensions to the existing redirection and command sequence operations
- several new built-in commands

These facilities are diagram shown in Figure 5.1.

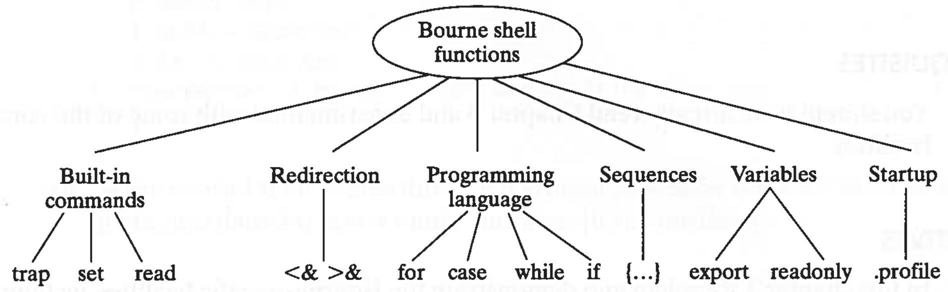

FIGURE 5.1

Bourne shell functionality.

START-UP

The Bourne shell is a regular C program whose executable file is stored as "/bin/sh". If your chosen shell is "/bin/sh," an interactive Bourne shell is invoked automatically when you log into UNIX. You may also invoke a Bourne shell manually from a script or from a terminal by using the command **sh** (which has several command-line options, described at the end of the chapter).

When an interactive Bourne shell is started, it searches for a file called ".profile" in the user's home directory. If it finds the file, it executes all of the shell commands that the file contains. Then, regardless of whether ".profile" was found, an interactive

Bourne shell displays its prompt and awaits user commands. The standard Bourne shell prompt is $, although it may be changed by setting the local variable PS1, described later in the chapter. (Note that noninteractive Bourne shells do not read any start-up files.)

One common use of ".profile" is to initialize environment variables such as TERM, which contains the type of your terminal, and PATH, which tells the shell where to search for executable files. Here's an example of a Bourne shell ".profile" startup file:

```
TERM=vt100                         # Set terminal type.
export TERM                        # Copy to environment.
# Set path and metacharacters
stty erase "^?" kill "^U" intr "^C" eof "^D"
PATH='.:$HOME/bin:/bin:/usr/sbin:/usr/bin:/usr/local/bin'
```

VARIABLES

The Bourne shell can perform the following variable-related operations:

- simple assignment and access
- testing of a variable for existence
- reading a variable from standard input
- making a variable read only
- exporting a local variable to the environment

The Bourne shell also defines several local and environment variables in addition to those mentioned in Chapter 4.

Creating/Assigning a Variable

The Bourne shell syntax for assigning a value to a variable is

```
{name=value}+
```

If a variable doesn't exist, it is implicitly created; otherwise, its previous value is overwritten. A newly created variable is always local, although it may be turned into an environment variable by means of a method I'll describe shortly. To assign a value that contains spaces, surround the value by quotes. Here's an example:

```
$ firstName=Graham lastName=Glass age=29  ...assign vars.
$ echo $firstName $lastName is $age
Graham Glass is 29                         ...simple access.
$ name=Graham Glass                        ...syntax error.
Glass: not found
$ name="Graham Glass"    ...use quotes to built strings.
$ echo $name                               ...now it works.
Graham Glass
$ _
```

Accessing a Variable

The Bourne shell supports the access methods shown in Figure 5.2. If a variable is accessed before it is assigned a value, it returns a null string.

Syntax	Action
$*name*	Replaced by the value of *name*.
${*name*}	Replaced by the value of *name*. This form is useful if the expression is immediately followed by an alphanumeric character that would otherwise be interpreted as part of the variable name.
${*name*−*word*}	Replaced by the value of *name* if set and *word* otherwise.
${*name*+*word*}	Replaced by *word* if *name* is set and nothing otherwise.
${*name*=*word*}	Assigns *word* to the variable *name* if *name* is not already set and then is replaced by the value of *name*.
${*name*?*word*}	Replaced by name if *name* is set. If *name* is not set, *word* is displayed to the standard error channel and the shell is exited. If *word* is omitted, then a standard error message is displayed instead.

FIGURE 5.2

Bourne shell special variables.

I personally find these techniques for accessing variables to be "hack" methods of dealing with certain conditions, so I hardly ever use them. However, it's good to be able to understand code that uses them. The following examples illustrate each access method. In the first example, I used braces to append a string to the value of a variable:

```
$ verb=sing                    ...assign a variable.
$ echo I like $verbing         ...there's no variable "verbing".
I like
$ echo I like ${verb}ing       ...now it works.
I like singing
$ _
```

Here's an example that uses command substitution to set the variable startDate to the current date if it's not already set to that:

```
$ startDate=${startDate-`date`}   ...if not set, run date.
$ echo $startDate                 ...look at its value.
Tue Wed 4 06:56:51 CST 1998
$ _
```

In the next example, I set the variable x to a default value and printed its value, all at the same time:

```
$ echo x = ${x=10}      ...assign a default value.
x = 10
$ echo $x               ...confirm the variable was set.
10
$ _
```

In the following example, I displayed messages on the basis of whether certain variables were set:

```
$ flag=1                                ...assign a variable.
$ echo ${flag+'flag is set'}            ...conditional message #1.
flag is set
$ echo ${flag2+'flag2 is set'}          ...conditional message #2.
                                        ...result is null
$ _
```

In the next example, I tried to access an undefined variable called grandTotal and received an error message instead:

```
$ total=10                              ...assign a variable.
$ value=${total?'total not set'}        ...accessed OK.
$ echo $value                                   ...look at its value.
10
$ value=${grandTotal?'grand total not set'}  ...not set.
grandTotal: grand total not set
$ _
```

In the final example, I ran a script that used the same access method as the previous example. Note that the script terminated when the access error occurred:

```
$ cat script.sh                         ...look at the script.
value=${grandTotal?'grand total is not set'}
echo done              # this line is never executed.
$ script.sh                             ...run the script.
script.sh: grandTotal: grand total is not set
$ _
```

Reading a Variable from Standard Input

The *read* command allows you to read variables from standard input. It works as shown in Figure 5.3. If you specify just one variable, the entire line is stored in the variable. Here's a sample script that prompts a user for his or her full name:

```
$ cat script.sh                         ...list the script.
echo "Please enter your name: \c"
read name                         # read just one variable.
```

```
echo your name is $name      # display the variable.
$ script.sh                         ...run the script.
Please enter your name: Graham Walker Glass
your name is Graham Walker Glass    ...whole line was read.
$ _
```

Shell Command: **read** {*variable*}+

read reads one line from standard input and then assigns successive words from the line to the specified variables. Any words that are left over are assigned to the last named variable.

FIGURE 5.3

Description of the *read* shell command.

Here's an example that illustrates what happens when you specify more than one variable:

```
$ cat script.sh                     ...list the script.
echo "Please enter your name: \c"
read firstName lastName      # read two variables.
echo your first name is $firstName
echo your last name is $lastName
$ script.sh                         ...run the script.
Please enter your name: Graham Walker Glass
your first name is Graham           ...first word.
your last name is Walker Glass      ...the rest.
$ script.sh                         ...run it again.
Please enter your name: Graham
your first name is Graham           ...first word.
your last name is                   ...only one.
$ _
```

Predefined Local Variables

In addition to the core predefined local variables, the Bourne shell defines the local variables shown in Figure 5.7. Here's a small shell script that illustrates the first three variables. In this example, the C compiler (**cc**) was invoked on a file that didn't exist, and therefore the system returned a failure exit code.

```
$ cat script.sh                     ...list the script.
echo there are $# command line arguments: $@
cc $1                   # compile the first argument.
echo the last exit value was $?    # display exit code.
$ script.sh nofile tmpfile          ...execute the script.
there are 2 command line arguments: nofile tmpfile
cc: Warning: File with unknown suffix (nofile) passed to ld
ld: nofile: No such file or directory
the last exit value was 4           ...cc errored.
$ _
```

Name	Value
$@	an individually quoted list of all the positional parameters
$#	the number of positional parameters
$?	the exit value of the last command
$!	the process ID of the last background command
$-	the current shell options assigned from the command line or by the built-in set command (see later)
$$	the process ID of the shell in use

FIGURE 5.7
Bourne predefined local variables.

The next example illustrates how $! may be used to kill the last background process:

```
$ sleep 1000 &                  ...create a background process.
29455                           ...process ID of background process.
$ kill $!                       ...kill it!
29455 Terminated
$ echo $!                       ...the process ID is still remembered.
29455
$ _
```

Predefined Environment Variables

In addition to the core predefined environment variables (listed in Chapter 2), the Bourne shell defines the environment variables shown in Figure 5.8. Here's a small example that illustrates the first three predefined environment variables. I set my prompt to something different by assigning a new value to PS1 and changed the delimiter character to a colon, saving the previous value in a local variable. Finally, I set PS2 to a new value and illustrated a situation in which the secondary prompt is displayed.

```
$ PS1="sh? "                    ...set a new primary prompt.
sh? oldIFS=$IFS                 ...remember the old value of IFS.
sh? IFS=":"                     ...change the word delimiter to a colon.
sh? ls:*.c                      ...this executes OK!
badguy.c    number.c    open.c      trunc.c     writer.c
fact2.c     number2.c   reader.c    who.c
sh? IFS=$oldIFS                        ...restore the old value of IFS.
sh? string="a long\                    ...assign a string over 2 lines
> string"                       ...">" is the secondary prompt.
sh? echo $string                ...look at the value of "string".
a long string
```

```
sh? PS2="??? "                ...change the secondary prompt.
sh? string="a long\           ...assign a long string.
??? string"                ..."???" is new secondary prompt.
sh? echo $string           ...look at the value of "string".
a long string
sh? _
```

Name	Value
$IFS	When the shell tokens a command line prior to its execution, it uses the characters in this variable as delimiters. IFS usually contains a space, a tab, and a newline character.
$PS1	This variable contains the value of the command-line prompt, $ by default. To change the prompt, simply set PS1 to a new value.
$PS2	This variable contains the value of the secondary command-line prompt (> by default) that is displayed when more input is required by the shell. To change the prompt, set PS2 to a new value.
$SHENV	If this variable is not set, the shell searches the user's home directory for the ".profile" start-up file when a new shell is created. If the variable is set, then the shell searches the directory specified by SHENV.

FIGURE 5.8
Bourne shell predefined environment variables.

ARITHMETIC

Although the Bourne shell itself doesn't support arithmetic directly, the **expr** utility does and is used as described in Figure 5.9. The following example illustrates some of the functions of **expr** and makes plentiful use of command substitution:

```
$ x=1                              ...initial value of x.
$ x=`expr $x + 1`                  ...increment x.
$ echo $x
2
$ x=`expr 2 + 3 \* 5`              ...* before +.
$ echo $x
17
$ echo `expr \( 2 + 3 \) \* 5`     ...regroup.
25
$ echo `expr length "cat"`         ...find length of "cat".
3
$ echo `expr substr "donkey" 4 3`  ...extract a substring.
key
```

```
$ echo `expr index "donkey" "ke"`         ...locate a substring.
4
$ echo `expr match "smalltalk" '.*lk'`...attempt a match.
9
$ echo `expr match "transputer" '*.lk'`      ...attempt match.
0
$ echo `expr "transputer" : '*.lk'`          ...attempt a match.
0
$ echo `expr \( 4 \> 5 \)`                   ...is 4 > 5 ?
0
$ echo `expr \( 4 \> 5 \) \| \( 6 \< 7 \)`   ...4>5 or 6<7?
1
$ _
```

Utility: **expr** *expression*

expr evaluates *expression* and sends the result to standard output. All of the components of *expression* must be separated by blanks, and all of the shell metacharacters must be escaped by a \. *expression* may yield a numeric or string result, depending on the operators that it contains. The result of *expression* may be assigned to a shell variable by the appropriate use of command substitution.

expression may be constructed by applying the following binary operators to integer operands, grouped in decreasing order of precedence:

OPERATOR	MEANING
* / %	multiplication, division, remainder
+ −	addition, subtraction
= > >= < <= !=	comparison operators
&	logical and
|	logical or

Parentheses may be used to explicitly control the order of evaluation. (They also must be escaped.) **expr** supports the following string operators as well:

OPERATOR	MEANING
string : regularExpression **match** *string regularExpression*	Both forms return the length of *string* if both sides match, and each returns zero otherwise.

FIGURE 5.9
Description of the **expr** command.

substr string start length	Returns the substring of *string*, starting from index *start* and consisting of *length* characters.
index string charList	Returns the index of the first character in *string* that appears in *charList*.
length string	Returns the length of *string*.

The format of *regularExpression* is defined in the appendix.

FIGURE 5.9 (*Continued*)

CONDITIONAL EXPRESSIONS

The control structures described in the next section often branch, depending on the value of a logical expression—that is, an expression that evaluates to true or false. The **test** utility supports a substantial set of UNIX-oriented expressions that are suitable for most occasions. It works as shown in Figure 5.10. A **test** expression may take the forms shown in Figure 5.11 **test** is very picky about the syntax of expressions; the spaces shown in this table are *not* optional. For examples of **test**, see the next section, which uses them in a natural context.

Utility: **test** *expression*

[*expression*] (equivalent form on some UNIX systems)

test returns a zero exit code if *expression* evaluates to true; otherwise, it returns a nonzero exit status. The exit status is typically used by shell control structures for branching purposes.

Some Bourne shells support **test** as a built-in command, in which case they support the second form of evaluation as well. The brackets of the second form must be surrounded by spaces in order to work.

(See the text for a description of the syntax of *expression*.)

FIGURE 5.10

Description of the **test** command.

Form	Meaning
-b *filename*	True if *filename* exists as a block special file.
-c *filename*	True if *filename* exists as a character special file.
-d *filename*	True if *filename* exists as a directory.
-f *filename*	True if *filename* exists as a nondirectory.
-g *filename*	True if *filename* exists as a "set group ID" file.
-h *filename*	True if *filename* exists as a symbolic link.
-k *filename*	True if *filename* exists and has its "sticky bit" set.
-l *string*	The length of *string*.
-n *string*	True if *string* contains at least one character.
-p *filename*	True if *filename* exists as a named pipe.
-r *filename*	True if *filename* exists as a readable file.
-s *filename*	True if *filename* contains at least one character.
-t *fd*	True if file descriptor *fd* is associated with a terminal.
-u *filename*	True if *filename* exists as a "set user ID" file.
-w *filename*	True if *filename* exists as a writable file.
-x *filename*	True if *filename* exists as an executable file.
-z *string*	True if *string* contains no characters.
str1 = *str2*	True if *str1* is equal to *str2*.
str1! = *str2*	True if *str1* is not equal to *str2*.
string	True if *string* is not null.
int1 -eq *int2*	True if integer *int1* is equal to integer *int2*.
int1 -ne *int2*	True if integer *int1* is not equal to integer *int2*.
int1 -gt *int2*	True if integer *int1* is greater than integer *int2*.
int1 -ge *int2*	True if integer *int1* is greater than or equal to integer *int2*.
int1 -lt *int2*	True if integer *int1* is less than integer *int2*.
int1 -le *int2*	True if integer *int1* is less than or equal to integer *int2*.
! *expr*	True if *expr* is false.
expr1 -a *expr2*	True if *expr1* and *expr2* are both true.
expr1 -o *expr2*	True if *expr1* or *expr2* are true.
\(*expr* \)	Escaped parentheses are used for grouping expressions.

FIGURE 5.11

test command expressions.

CONTROL STRUCTURES

The Bourne shell supports a wide range of control structures that make the shell suitable as a high-level programming tool. Shell programs are usually stored in scripts and are commonly used to automate maintenance and installation tasks. The next few subsections describe the control structures in alphabetical order; they assume that you are already familiar with at least one high-level programming language.

case .. in .. esac

The case command supports multiway branching based on the value of a single string, It has the syntax described in Figure 5.12. Here's an example of a script called "menu.sh" that makes use of a *case* control structure (this script is also available online; see the preface for information):

```
#! /bin/sh
echo menu test program
stop=0                      # reset loop termination flag.
while test $stop -eq 0      # loop until done.
do
 cat << ENDOFMENU             # display menu.
 1    : print the date.
 2, 3: print the current working directory.
 4    : exit
ENDOFMENU
 echo
 echo 'your choice? \c'     # prompt.
 read reply                 # read response.
 echo
 case $reply in             # process response.
   "1")
     date                   # display date.
     ;;
   "2"|"3")
     pwd                    # display working directory.
     ;;
   "4")
     stop=1                 # set loop termination flag.
     ;;
   *)                       # default.
     echo illegal choice    # error.
     ;;
 esac
done
```

case *expression* **in**
pattern { | *pattern* } *)
list
;;
esac

expression is an expression that evaluates to a string, *pattern* may include wildcards, and *list* is a list of one or more shell commands. You may include as many pattern–list associations as you wish. The shell evaluates *expression* and then compares it with each pattern in turn, from top to bottom. When the first matching pattern is found, its associated list of commands is executed, and then the shell skips to the matching **esac**. A series of patterns separated by "or" symbols (|) is all associated with the same list. If no match is found, then the shell skips to the matching **esac**.

FIGURE 5.12
Description of the *case* shell command.

Here's the output from the "menu.sh" script:

```
$ menu.sh
menu test program
  1    : print the date.
  2, 3: print the current working directory.
  4    : exit
your choice? 1
Thu Feb  5 07:09:13 CST 1998
  1    : print the date.
  2, 3: print the current working directory.
  4    : exit
your choice? 2
/home/glass
  1    : print the date.
  2, 3: print the current working directory.
  4    : exit
your choice? 5
illegal choice
  1    : print the date.
  2, 3: print the current working directory.
  4    : exit
your choice? 4
$ _
```

`for .. do .. done`

The *for* command allows a list of commands to be executed several times, using a different value of the loop variable during each iteration. Its syntax is shown in Figure 5.13. Here's an example of a script that uses a *for* control structure:

```
$ cat for.sh                    ...list the script.
for color in red yellow green blue
do
 echo one color is $color
done
$ for.sh                        ...execute the script.
one color is red
one color is yellow
one color is green
one color is blue
$ _
```

for *name* [**in** {*word*}*]
do
  list
done

The **for** command loops the value of the variable *name* through each *word* in the word list, evaluating the commands in *list* after each iteration. If no word list is supplied, $@ ($1..) is used instead. A *break* command causes the loop to end immediately, and a *continue* command causes the loop to jump immediately to the next iteration.

FIGURE 5.13

Description of the *for* shell command.

`if .. then .. fi`

The *if* command supports nested conditional branches. It has the syntax shown in Figure 5.14. Here's an example of a script that uses an *if* control structure:

```
$ cat if.sh                         ...list the script.
echo 'enter a number: \c'
read number
if [ $number -lt 0 ]
then
 echo negative
elif [ $number -eq 0 ]
then
 echo zero
else
 echo positive
fi
$ if.sh                                           ...run the script.
enter a number: 1
positive
$ if.sh                                           ...run the script again.
enter a number: -1
negative
$ _
```

if *list1*
then
  list2
elif *list3*   . . . optional, **elif** part may be repeated several times.
then
  list4
else   . . . optional, **else** part may occur zero or one times.
  list5
fi

The commands in *list1* are executed. If the last command in *list1* succeeds, the commands in *list2* are executed. If the last command in *list1* fails and there are one or more *elif* components, then a successful command list following an *elif* causes the commands following the associated *then* to be executed. If no successful lists are found and there is an *else* component, the commands following the *else* are executed.

FIGURE 5.14

Description of the *if* shell command.

until .. do .. done

The *until* command executes one series of commands as long as another series of commands fails. It has the syntax shown in Figure 5.16. Here's an example of a script that uses an *until* control structure:

```
$ cat until.sh                          ...list the script.
x=1
until [ $x -gt 3 ]
do
 echo x = $x
```

```
 x=`expr $x + 1`
done
$ until.sh                       ...execute the script.
x = 1
x = 2
x = 3
$ _
```

until *list1*
do
 list2
done

The *until* command executes the commands in *list1* and ends if the last command in *list1* succeeds; otherwise, the commands in *list2* are executed and the process is repeated. If *list2* is empty, the *do* keyword should be omitted. A *break* command causes the loop to end immediately, and a *continue* command causes the loop to jump immediately to the next iteration.

FIGURE 5.16

Description of the *until* shell command.

while .. done

The *while* command executes one series of commands as long as another series of commands succeeds. Its syntax is shown in Figure 5.17. Here's an example of a script that uses a *while* control structure to generate a small multiplication table:

```
$ cat multi.sh                          ...list the script.
if [ "$1" -eq "" ]; then
   echo "Usage: multi number"
   exit
fi
x=1                                     # set outer loop value
while [ $x -le $1 ]                     # outer loop
do
  y=1                            # set inner loop value
  while [ $y -le $1 ]
  do                                    # generate one table entry
    echo `expr $x \* $y` "  \c"
    y=`expr $y + 1`                     # update inner loop count
```

```
        done
        echo                                    # blank line
        x=`expr $x + 1`                     # update outer loop count
    done
    $ multi.sh 7                            ...execute the script.
    1       2       3       4       5       6       7
    2       4       6       8       10      12      14
    3       6       9       12      15      18      21
    4       8       12      16      20      24      28
    5       10      15      20      25      30      35
    6       12      18      24      30      36      42
    7       14      21      28      35      42      49
    $ _
```

while *list1*
do
  list2
done

The *while* command executes the commands in *list1* and ends if the last command in *list1* fails; otherwise, the commands in *list2* are executed and the process is repeated. If *list2* is empty, the *do* keyword should be omitted. A *break* command causes the loop to end immediately, and a *continue* command causes the loop to jump immediately to the next iteration.

FIGURE 5.17

Description of the *while* shell command.