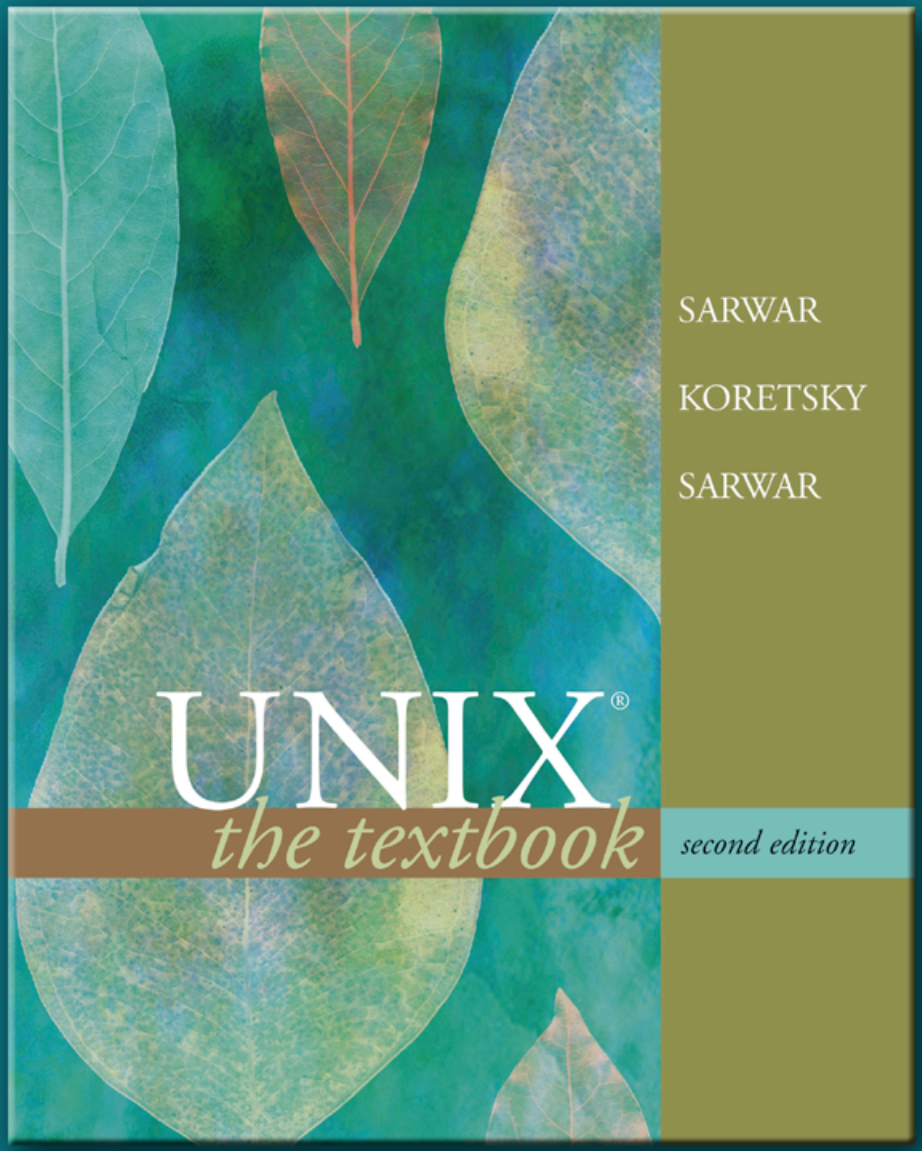

Chapter 15

Introductory Bourne Shell Programming

PEARSON Addison Wesley

Objectives

- To introduce the concept of shell programming
- To discuss how shell programs are executed
- To describe the concept and use of shell variables
- To discuss how command line arguments are passed to shell programs
- To explain the concept of command substitution
- To describe some basic coding principles
- To write and discuss some shell scripts
- To cover the commands and primitives

Introduction

- **Shell script**: a shell program, consists of shell commands to be executed by a shell and is stored in ordinary UNIX file
- **Shell variable:** read/write storage place for users and programmers to use as a scratch pad for completing a task
- **Program control flow commands (or statements):** allow non sequential execution of commands in a shell script and repeated execution of a block of commands

Running a Bourne Shell Script

- Ways of running a Bourne Shell
  - Make the script file executable by adding the execute permission to the existing access permissions for the file

    ```
    $ chmod u+x script_file
    $
    ```

  - Run the /bin/sh command with the script file as its parameter

    ```
    $ /bin/sh script_file
    $
    ```

  - Force the current shell to execute a script in the Bourne shell, regardless of your current shell

    ```
    #!/bin/sh
    ```

- Null command (:)

  When the C shell reads : as the first character it returns a Bourne shell process that executes the commands in the script. The : command returns true

Shell Variables

TABLE 15.1 Some Important Writable Bourne Shell Environment Variables

Environment Variable	Purpose of the Variable
CDPATH	Contains directory names that are searched, one by one, by the `cd` command to find the directory passed to it as a parameter; the `cd` command searches the current directory if this variable is not set
EDITOR	Default editor used in programs such as the e-mail program
ENV	Path along which UNIX looks to find configuration files
HOME	Name of home directory, when user first logs on
MAIL	Name of user's system mailbox file
MAILCHECK	How often (in seconds) the shell should check user's mailbox for new mail and inform user accordingly
PATH	Variable that contains user's search path—the directories that a shell searches to find an external command or program
PPID	Process ID of the parent process
PS1	Primary shell prompt that appears on the command line, usually set to `$`
PS2	Secondary shell prompt displayed on second line of a command if shell thinks that the command is not finished, typically when the command terminates with a backslash (`\`), the escape character
PWD	Name of the current working directory
TERM	Type of user's console terminal

Read-only Shell Variables

TABLE 15.2 Some Important Read-Only Bourne Shell Environment Variables

Environment Variable	Purpose of the Variable
$0	Name of program
$1–$9	Values of command line arguments 1 through 9
*$**	Values of all command line arguments
$@	Values of all command line arguments; each argument individually quoted if `$@` is enclosed in quotes, as in `"$@"`
$#	Total number of command line arguments
$$	Process ID (PID) of current process
$?	Exit status of most recent command
$!	PID of most recent background process

Display Shell Variables

```
$ set
AUTHSTATE=compat
CGI_DIRECTORY=/usr/lpp/internet/server_root/cgi-bin
ERRNO=10
FCEDIT=/usr/bin/ed
HOME=/home/inst/msarwar
IFS='
'
LOCPATH=/usr/lib/nls/loc
LOGIN=msarwar
LOGNAME=msarwar
MAIL=/usr/spool/mail/msarwar
MAILCHECK=600
MAILMSG='[YOU HAVE NEW MAIL]'
PATH=/usr/lpp/workbench/bin:/usr/lpp/Java/bin:/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/loca
    l/bin:/usr/local/share/bin:/home/inst/msarwar/bin:/usr/bin/X11:/sbin:.
PPID=18632
PS1='$ '
PS2='> '
PS3='#? '
PS4='+ '
PWD=/home/inst/msarwar
SHELL=/usr/bin/ksh
TERM=vt100
TMOUT=0
TZ=PST8PDT
USER=msarwar
$
```

Display Shell Variables

```
$ env
LANG=en_US
LOGIN=msarwar
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
IMQCONFIGCL=/etc/IMNSearch/dbcshelp
PATH=/usr/lpp/workbench/bin:/usr/lpp/Java/bin:/usr/bin:/etc:/usr/sbin:/usr/ucb:/us
   r/local/bin:/usr/local/share/
bin:/home/inst/msarwar/bin:/usr/bin/X11:/sbin:.
IMQCONFIGSRV=/etc/IMNSearch
CGI_DIRECTORY=/usr/lpp/internet/server_root/cgi-bin
LOGNAME=msarwar
MAIL=/usr/spool/mail/msarwar
LOCPATH=/usr/lib/nls/loc
USER=msarwar
DOCUMENT_SERVER_MACHINE_NAME=pccaix
AUTHSTATE=compat
SHELL=/usr/bin/ksh
ODMDIR=/etc/objrepos
DOCUMENT_SERVER_PORT=80
HOME=/home/inst/msarwar
TERM=vt100
MAILMSG=[YOU HAVE NEW MAIL]
PWD=/home/inst/msarwar
DOCUMENT_DIRECTORY=/usr/lpp/internet/server_root/pub
TZ=PST8PDT
$
```

IN-CHAPTER EXERCISES

15.1. Display the names and values of all the shell variables on your UNIX machine. What command(s) did you use?

15.2. Create a file that contains a shell script comprising the `date` and `who` commands, one on each line. Make the file executable and run the shell script. List all the steps for completing this task.

Reading and Writing Shell Variables

```
variable1=value1[ariable2=value2…variableN=valueN]
```

Purpose:

Assign values ‘value1,…,valueN’ to ‘variable1,…, variableN’ respectively –no space allowed before and after the equals sign

Reading and Writing Shell Variables

```
$ name=John
$ echo $name
John
$ name=John Doe
Doe: not found
$ name="John Doe"
$ echo $name
John Doe
$ name=John*
$ echo $name
John.Bates.letter John.Johnsen.memo John.email
$ echo "$name"
John*
$ echo "The name $name sounds familiar!"
The name John* sounds familiar!
$ echo \$name
$name
$ echo '$name'
$name
$
```

Command Substitution

- **Command Substitution:** When a command is enclosed in back quotes, the shell executes the command and substitutes the command (including back quotes) with the output of the command
- `` `command` ``

  Purpose: Substitute its output for `` `command` ``

Command Substitution

```
$ command=pwd
$ echo "The value of command is: $command."
The value of command is: pwd.
$ command=`pwd`
$ echo "The value of command is: $command."
The value of command is: /users/faculty/sarwar/unixbook/examples.
$
$ echo "The date and time is `date`."
The date and time is Fri May 7 13:26:42 PDT 2004.
$
```

IN-CHAPTER EXERCISES

15.3. Assign your full name to a shell variable called *myname* and echo its value. How did you accomplish the task? Show your work.

15.4. Assign the output of `echo "Hello, world!"` command to the *myname* variable and then display the value of *myname*. List the commands that you executed to complete your work.

Exporting Environment

`export [name-list]`

Purpose Export the names and copies of the current values in the 'name-list' to every command executed from this point on

```
$ name="John Doe"
$ export name
$
$ cat display_name
echo $name
exit 0
$
$ name="John Doe"
$ display_name

$
$ name="John Doe"
$ export name
$ display_name
John Doe
$ echo $?
0
$
```

```
$ cat export_demo
#!/bin/sh
name="John Doe"
export name
display_change_name
display_name
exit 0
$ cat display_change_name
#!/bin/sh
echo $name
name="Plain Jane"
echo $name
exit 0
$ export_demo
John Doe
Plain Jane
John Doe
$
```

Resetting Variables

`unset [name-list]`

Purpose Reset or remove the variable or function corresponding to the names in ‘name-list’, where ‘name-list’ is a list of names separated by spaces

```
$ name=John place=Corvallis
$ echo "$name $place"
John Corvallis
$ unset name
$ echo "$name"

$ echo "$place"
Corvallis
$
$ unset name place
$
```

```
$ country=
$ echo "$country"

$
```

Creating Read-Only Defined Variables

`readonly [name-list]`

Purpose Prevent assignment of new values to the variables in 'name-list'

```
$ name=Jim
$ place=Ames
$ readonly name place
$ echo "$name $place"
John Ames
$ name=Art place="Ann Arbor"
place: is read only
$ name=Art
name: is read only
$

$ readonly
LOGNAME=msarwar
name=Jim
place=Ames
$
```

Reading from Standard Input

`read variable-list`

Purpose Read one line from standard input and assign words in the line to variables in 'name-list'

```
$ cat read_demo
#!/bin/sh
echo "Enter input: \c"
read line
echo "You entered: $line"
echo "Enter another line: \c"
read word1 word2 word3
echo "The first word is: $word1"
echo "The second word is: $word2"
echo "The rest of the line is: $word3"
exit 0
$
```

```
$ read_demo
Enter input: UNIX rules the network computing world!
You entered: UNIX rules the network computing world!
Enter another line: UNIX rules the network computing world!
The first word is: UNIX
The second word is: rules
The rest of the line is: the network computing world!
$
```

IN-CHAPTER EXERCISES

15. 5. Give commands for reading a value into the *myname* variable from the keyboard and exporting it so that commands executed in any child shell have access to the variable.

15.6. Copy the value *myname* variable to another variable, *anyname*. Make the *anyname* variable readonly and **unset** both the *myname* and *anyname* variables. What happened? Show all your work.

Special Characters for the `echo` Command

TABLE 15.3 Special Characters for the `echo` Command*

Character	Meaning
`\b`	Backspace
`\c`	Prints line without moving cursor to next line
`\f`	Form feed
`\n`	Newline (move cursor to next line)
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash (escape special meaning of backslash)
`\0N`	Character whose ASCII number is octal *N*

* You might have to use \\ instead of \ on some UNIX systems.

Passing Arguments to Shell Scripts

`shift[N]`

Purpose Shift the command line arguments *N* positions to the left

`set [options] [argument-list]`

Purpose Set values of the positional arguments to the arguments in 'argument-list' when executed without an argument, the **set** command displays the names of all shell variables and their current values

Passing Arguments to Shell Scripts

```
$ cat cmdargs_demo
#!/bin/sh
echo "The command name is: $0."
echo "The number of command line arguments passed as parameters are $#."
echo "The value of the command line arguments are: $1 $2 $3 $4 $5 $6 $7 $8 $9."
echo "Another way to display values of all of the arguments: $@."
echo "Yet another way is: $*."
exit 0
$ cmdargs_demo a b c d e f g h i
The command name is: cmdargs_demo.
The number of command line arguments passed as parameters are 9.
The value of the command line arguments are: a b c d e f g h i.
Another way to display values of all of the arguments: a b c d e f g h i.
Yet another way is: a b c d e f g h i.
$ cmdargs_demo One Two 3 Four 5 6
The command name is: cmdargs_demo.
The number of command line arguments passed as parameters are 6.
The value of the command line arguments are: One Two 3 Four 5 6 .
Another way to display values of all of the arguments: One Two 3 Four 5 6.
Yet another way is: One Two 3 Four 5 6.
$
```

Passing Arguments to Shell Scripts

```
$ cat shift_demo
#!/bin/sh
echo "The program name is $0."
echo "The arguments are: $@"
echo "The first three arguments are: $1 $2 $3"
shift
echo "The program name is $0."
echo "The arguments are: $@"
echo "The first three arguments are: $1 $2 $3"
shift 3
echo "The program name is $0."
echo "The arguments are: $@"
echo "The first three arguments are: $1 $2 $3"
exit 0
$ shift_demo 1 2 3 4 5 6 7 8 9 10 11 12
The program name is shift_demo.
The arguments are: 1 2 3 4 5 6 7 8 9 10 11 12
The first three arguments are: 1 2 3
The program name is shift_demo.
The arguments are: 2 3 4 5 6 7 8 9 10 11 12
The first three arguments are: 2 3 4
The program name is shift_demo.
The arguments are: 5 6 7 8 9 10 11 12
The first three arguments are: 5 6 7
$
```

Passing Arguments to Shell Scripts

```
$ date
Fri May 7 13:26:42 PDT 2004
$ set `date`
$ echo "$@"
Fri May 7 13:26:42 PDT 2004
$ echo "$2 $3, $6"
May 7, 2004
$
```

```
$ cat set_demo
#!/bin/sh
filename="$1"
set `ls -il $filename`
inode="$1"
size="$6"
echo "Name\tInode\tSize"
echo
echo "$filename\t$inode\t$size"
exit 0
$ set_demo lab3
Name    Inode      Size

lab3    856110     591
$
```

IN-CHAPTER EXERCISES

15.7. Write a shell script that displays all command line arguments, shifts them to the left by two positions and redisplays them. Show the script along with a few sample runs.

15.8. Update the shell script in Exercise 15.7 so that, after accomplishing the original task, it sets the positional arguments to the output of the `who | head -1` command and displays the positional arguments again.

Comments and Program Headers

- Put comments in your programs to describe the purpose of a particular set of commands
- Use program header for every shell script you write, including:
  - Name of the file containing the script
  - Name of the author
  - Date written
  - Date last modified
  - Purpose of the script
  - A brief description of the algorithm used to implement the solution to the problem at hand

```
# File Name:              ~/Bourne/examples/set_demo
# Author:                 Syed Mansoor Sarwar
# Date Written:           August 10, 1999
# Date Last Modified:     August 10, 1999
# Purpose:                To illustrate how the set command works
# Brief Description:      The script runs with a filename as the only command
#                         line argument, saves the filename, runs the set
#                         command to assign output of ls -il command to
#                         positional arguments ($1 through $9), and displays
#                         file name, its inode number, and its size in bytes.
```

Program Control Flow Commands

- Used to determine the sequence in which statements in a shell script execute
- The three basic types of statements for controlling flow of a script are:
  - Two-way branching
  - Multiway branching
  - Repetitive execution of one or more commands

Program Control Flow Commands

SYNTAX

```
if expression
    then
       [elif expression
       then
          then-command-list]
       ...
       [else
          else-command-list]
fi
```

Purpose: To implement two-way or multiway branching

SYNTAX

```
if expression
    then
         then-commands
fi
```

Purpose: To implement two-way branching

SYNTAX

```
test [ expression ]
Or
[[ expression ]]
```

Purpose: To evaluate 'expression' and return true or false status

Program Control Flow Commands

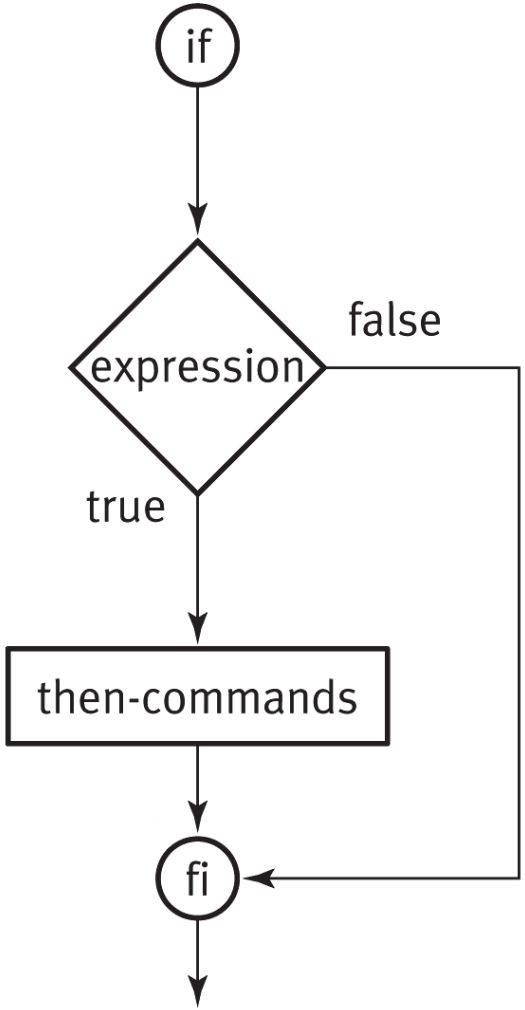

Figure 15.1 Semantics of the `if-then-fi` statement

Operators for the `test` Command

TABLE 15.4 Operators for the `test` Command

File Testing		Integer Testing		String Testing	
Expression	**Return Value**	**Expression**	**Return Value**	**Expression**	**Return Value**
-d file	True if 'file' is a directory	**int1 –eq int2**	True if 'int1'and 'int2'are equal	**str**	True if 'str' is not an empty string
-f file	True if 'file' is an ordinary file	**int1 -ge int2**	True if 'int1' is greater than or equal to 'int2'	**str1 = str2**	True if 'str1'and 'str2' are the same
-r file	True if 'file' is readable	**int1 -gt int2**	True if 'int1' is greater than 'int2'	**str1 != str2**	True if 'str1' and 'str2' are not the same
-s file	True if length of 'file' is nonzero	**int1 –le int2**	True if 'int1' is less than or equal to 'int2'	**-n str**	True if the length of 'str' is greater than zero
-t [filedes]	True if file descriptor 'filedesis' associated with the terminal	**int1 –lt int2**	True if 'int1' is less than 'int2'	**-z** str	True if the length of 'str' is zero
-w file	True if 'file' is writable	**int1 –ne int2**	True if 'int1' is not equal to 'int2'		
-x file	True if 'file' is executable				

```
$ cat if_demo1
#!/bin/sh
if test $# -eq 0
     then
          echo "Usage: $0 ordinary_file"
          exit 1
fi
if test $# -gt 1
     then
          echo "Usage: $0 ordinary_file"
          exit 1
fi
if test -f "$1"
     then
          filename="$1"
          set `ls -il $filename`
          inode="$1"
          size="$6"
          echo "Name\tInode\tSize"
          echo
          echo "$filename\t$inode\t$size"
          exit 0
fi
echo "$0: argument must be an ordinary file"
exit 1
```

```
$ if_demo1
Usage: if_demo1 ordinary_file
$ if_demo1 lab3 lab4
Usage: if_demo1 ordinary_file
$ if_demo1 dir1
if_demo1: argument must be an ordinary file
$ if_demo1 lab3
Name      Inode    Size

lab3    856110    591
$
```

Program Control Flow Commands

```
if expression
    then
       then-command
    else
       else-command
fi
```

SYNTAX

Purpose: To implement two-way branching

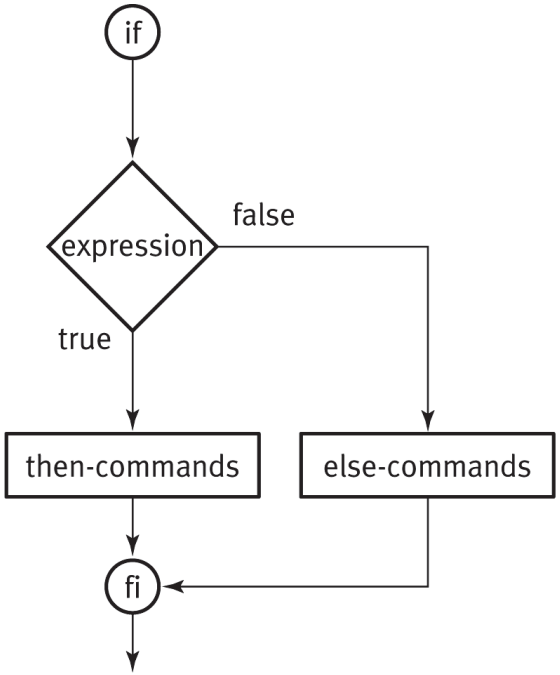

Figure 15.2 Semantics of the `if-then-else-fi` statement

Example Script

```
$ cat if_demo2
#!/bin/sh
if [ $# -eq 0 ]
   then
      echo "Usage: $0 ordinary_file"
      exit 1
fi
if [ $# -gt 1 ]
   then
      echo "Usage: $0 ordinary_file"
      exit 1
fi
if [ -f "$1" ]
   then
      filename="$1"
      set `ls -il $filename`
      inode="$1"
      size="$6"
      echo "Name\tInode\tSize"
      echo
      echo "$filename\t$inode\t$size"
      exit 0
   else
      echo "$0: argument must be an ordinary file"
      exit 1
fi
$
```

Program Control Flow Commands

SYNTAX

```
if expression1
  then
     then-commands
  elif expression2
     elif1-commands
  elif expression3
     elif2-commands
  ...
  else
     else-command
fi
```

Purpose: To implement multiway branching

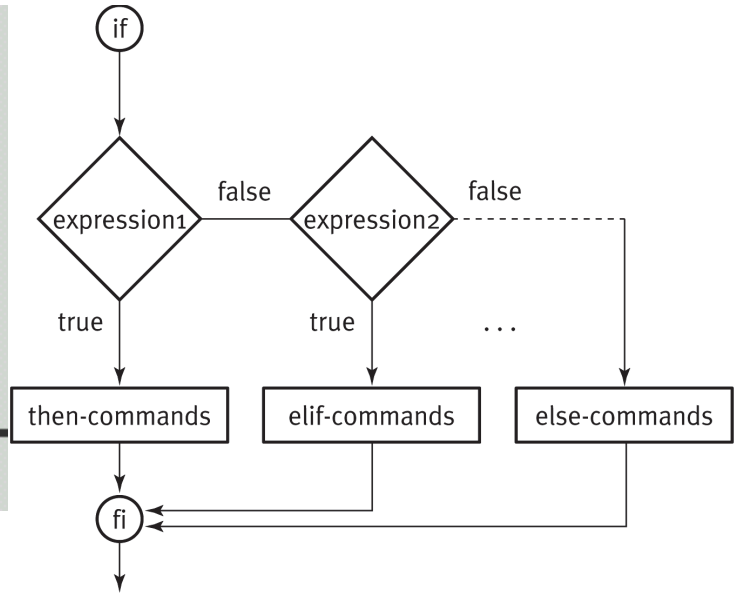

Figure 15.3 Semantics of the `if-then-elif-else-fi` statement

Example Script

```
$ cat if_demo3
#!/bin/sh
if [ $# -eq 0 ]
     then
          echo "Usage: $0 file"
          exit 1
     elif [ $# -gt 1 ]
     then
          echo "Usage: $0 file"
          exit 1
     elif [ -d "$1" ]
          then
               nfiles=`ls "$1" | wc -w`
               echo "The number of files
               exit 0
fi
```

```
else
     ls "$1" 2> /dev/null | grep "$1" 2> /dev/null 1>&2
     if [ $? -ne 0 ]
     then
          echo "$1: not found"
          exit 1
     fi
     if [ -f "$1" ]
          then
               filename="$1"
               set `ls -il $filename`
               # Please see the warning at the end of section 15.4
               shift 4
               inode="$1"
               size="$6"
               echo "Name\tInode\tSize"
               echo
               echo "$filename\t$inode\t$size"
               exit 0
          else
               echo "$0: argument must be an ordinary file or directory"
               exit 1
     fi
```

Program Control Flow Commands

```
$ if_demo3 /bin/ls
Name      Inode     Size

/bin/ls   50638     18844
$ if_demo3 file1
file1: not found
$ if_demo3 dir1
The number of files in the directory is 4
$ if_demo3 lab3
Name      Inode          Size

lab3      856110         591
$
```

IN-CHAPTER EXERCISES

15.9. Create the if_demo2 script file and run it with no argument, more than one argument, and one argument only. While running the script with one argument, use a directory as the argument. What happens? Does the output make sense?

15.10. Write a shell script whose single command line argument is a file. If you run the program with an ordinary file, the program displays the owner's name and last update time for the file. If the program is run with more than one argument, it generates meaningful error messages.

The `for` Statement

SYNTAX

```
for variable [in argument-list]
do
   command-list
done
```

Purpose: To execute commands in 'command-list' as many times as the number of words in the 'argument-list'; without the optional part, 'in argument-list', the arguments are supplied at the command

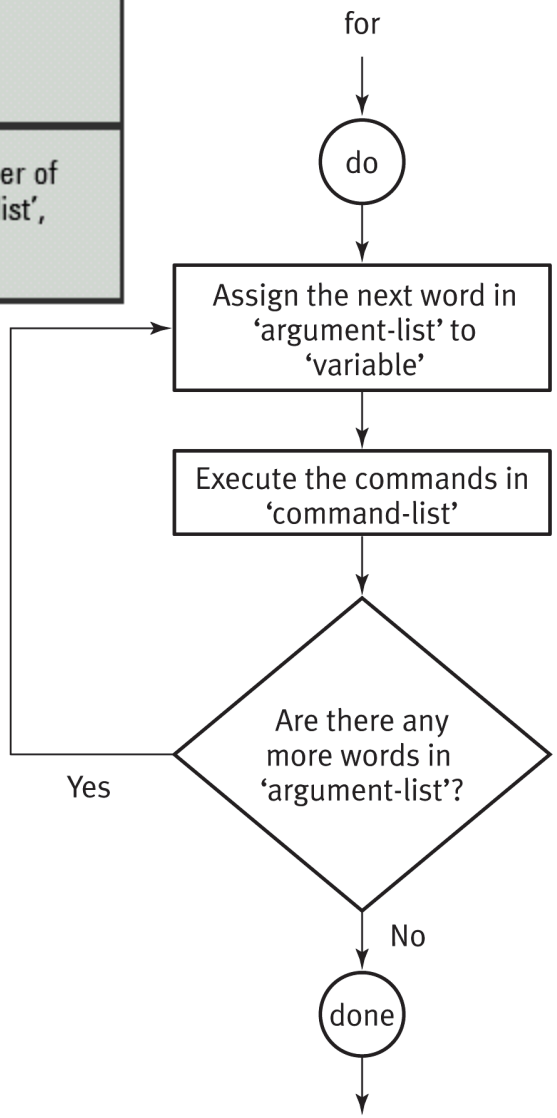

Figure 15.4 Semantics of the `for` statement

The `for` Statement

```
$ cat for_demo1
#!/bin/sh
for people in Debbie Jamie John Kitty Kuhn Shah
do
   echo "$people"
done
exit 0
$ for_demo1
Debbie
Jamie
John
Kitty
Kuhn
Shah
$
```

The `for` Statement

```
$ cat user_info
#!/bin/sh
for user
do
# Don't display anything if a login name is not found in /etc/passwd
grep "^"`echo $user":"` /etc/passwd 1> /dev/null 2>&1
if [ $? -eq 0 ]
    echo "$user:\t\c"
    grep "^"`echo $user":"` /etc/passwd | cut -f5 -d':'
fi
done
exit 0
$ user_info dheckman ghacker msarwar
dheckman:  Dennis R. Heckman
ghacker:   George Hacker
msarwar:   Mansoor Sarwar
$
```

The `while` statement

SYNTAX

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

Purpose: To execute commands in 'command-list' as long as 'expression' evaluates to true

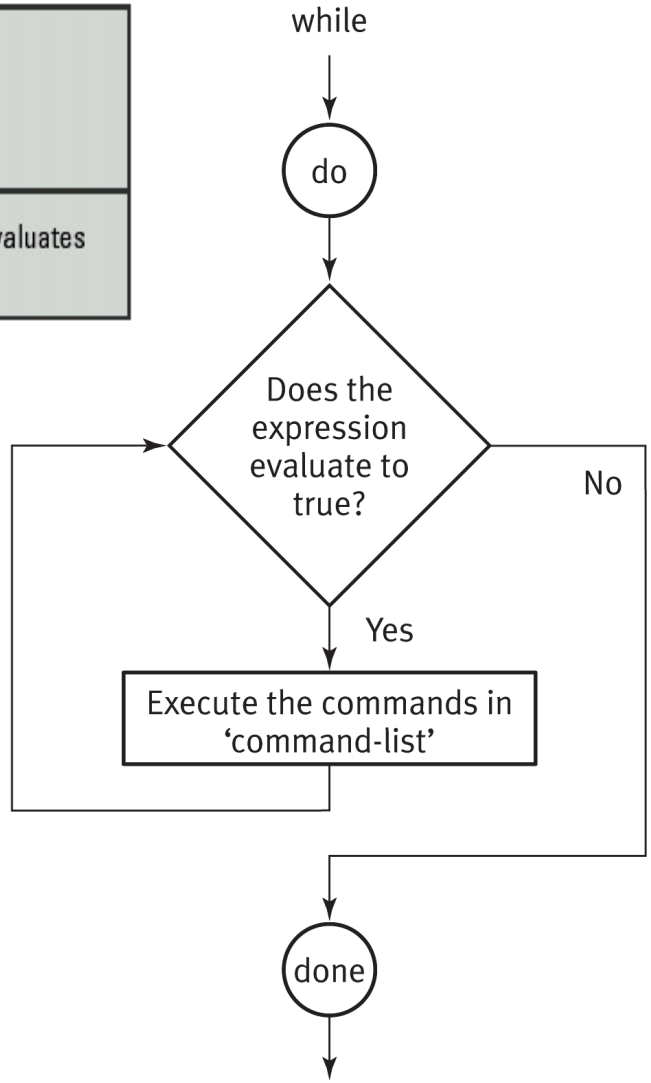

Figure 15.5 Semantics of the `while` statement

The `while` statement

```
$ cat while_demo
#!/bin/sh
secretcode=agent007
echo "Guess the code!"
echo "Enter your guess: \c"
read yourguess
while [ "$secretcode" != "$yourguess" ]
do
echo "Good guess but wrong. Try again!"
echo "Enter your guess: \c"
read yourguess
done
echo "Wow! You are a genius!!"
exit 0
$ while_demo
Guess the code!
Enter your guess: star wars
Good guess but wrong. Try again!
Enter your guess: columbo
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!!
$
```

The `until` Statement

SYNTAX

```
until expression
do
   command-list
done
```

Purpose: To execute commands in 'command-list' as long as 'expression' evaluates to false

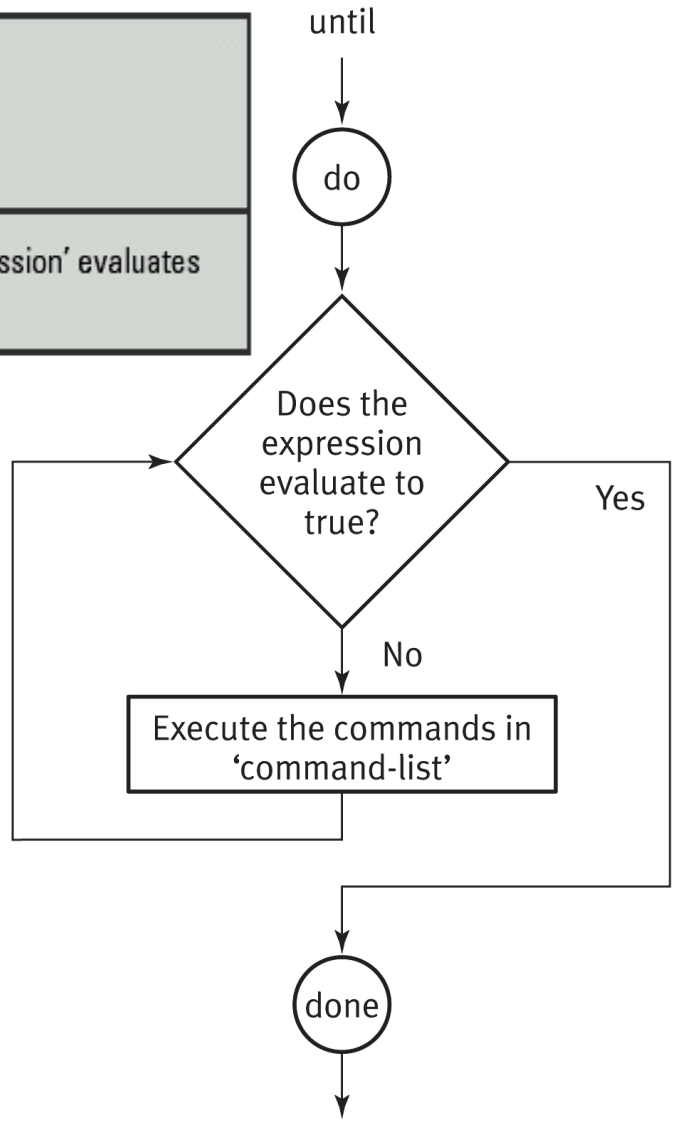

Figure 15.6 Semantics of the `until` statement

The `until` Statement

```
$ cat until_demo
#!/bin/sh
secretcode=agent007
echo "Guess the code!"
echo "Enter your guess: \c"
read yourguess
until [ "$secretcode" = "$yourguess" ]
do
     echo "Good guess but wrong. Try again!"
     echo "Enter your guess: \c"
     read yourguess
done
echo "Wow! You are a genius!!"
exit 0
```

```
$ until_demo
Guess the code!
Enter your guess: Inspector Gadget
Good guess but wrong. Try again!
Enter your guess: Peter Sellers
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!!
$
```

The `break` and `continue` Statements

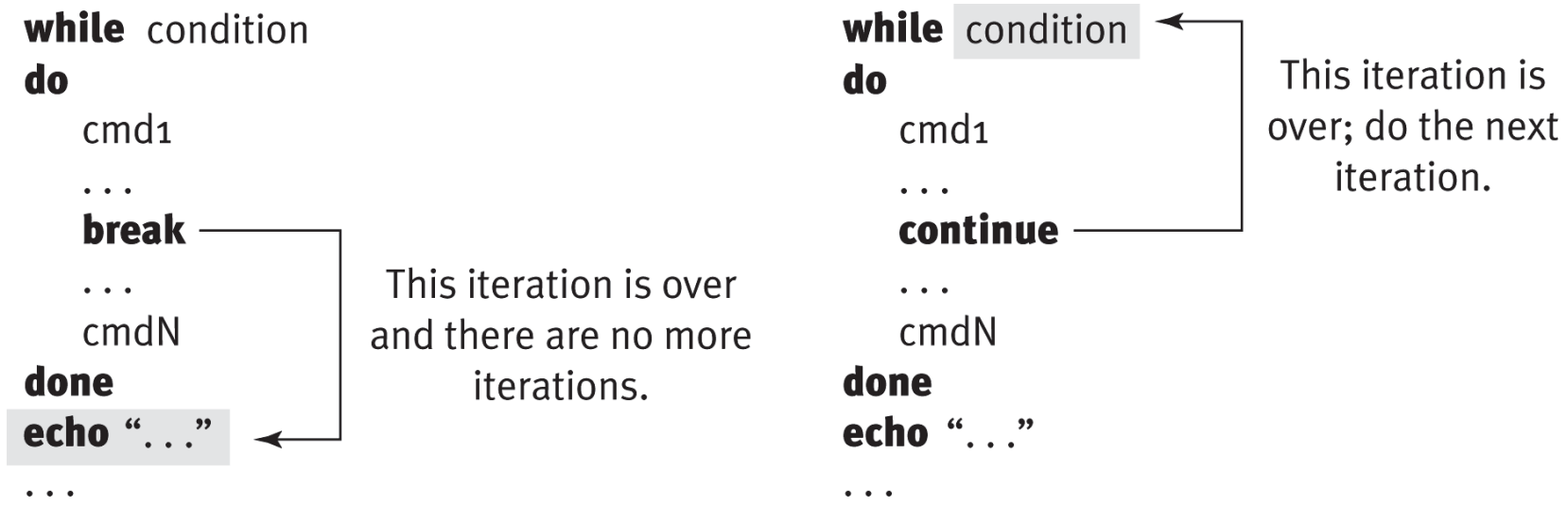

Figure 15.7 Semantics of the `break` and `continue` commands

IN-CHAPTER EXERCISES

15.11. Write a shell script that takes a list of host names on your network as command line arguments and displays whether the hosts are up or down. Use the `ping` command to display the status of a host and the `for` statement to process all host names.

15.12. Rewrite the script in Exercise 15.11, using the `while` statement. Rewrite it again, using the `until` statement.

The `case` Statement

```
case test-string in
   pattern1)   command-list1
               ;
   pattern2)   command-list2
               ;;
   ...
   patternN)   command-listN
               ;;
esac
```

SYNTAX

Purpose: To implement multiway branching like a nested `if`

The `case` Statement

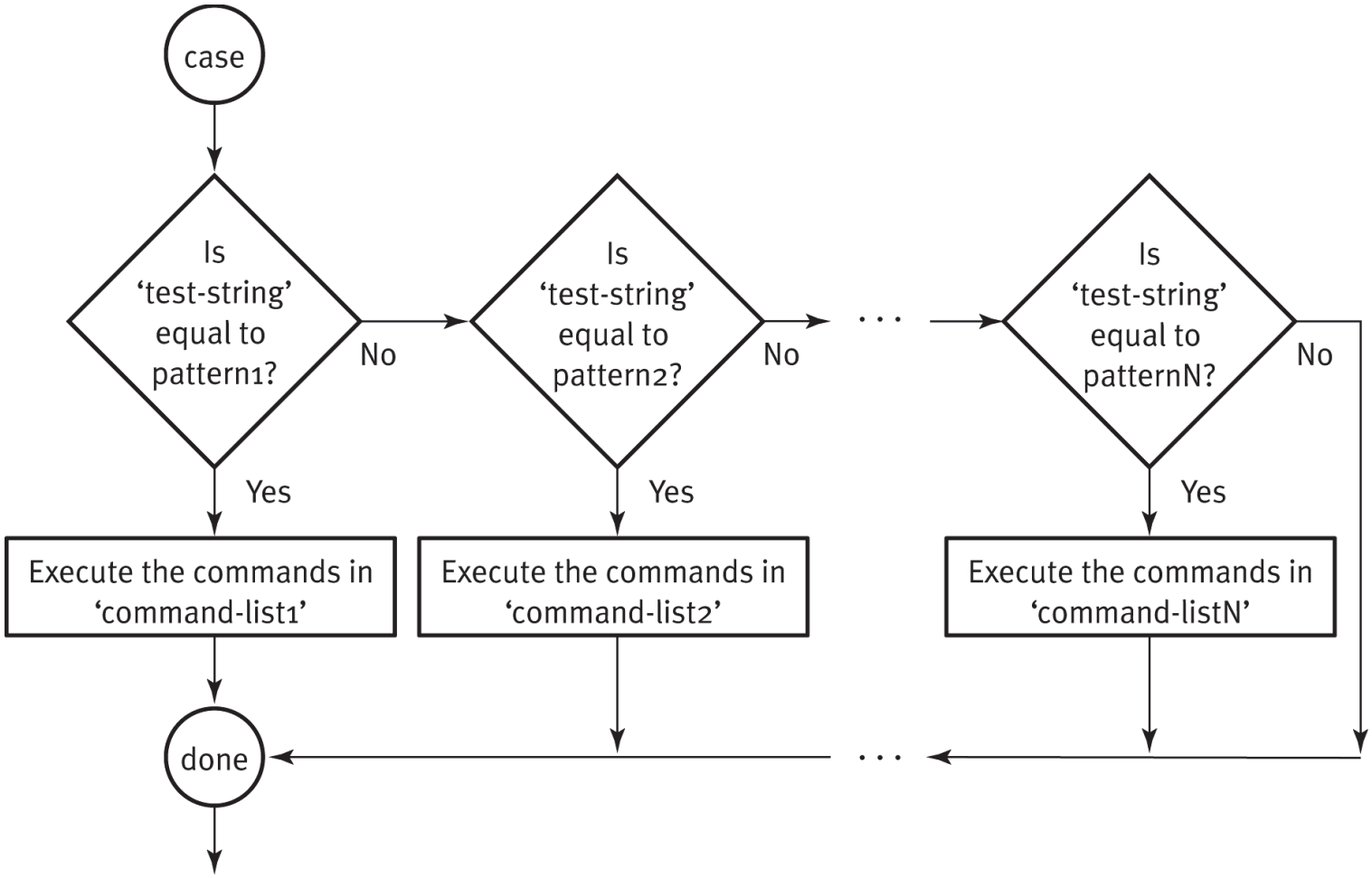

Figure 15.8 Semantics of the `case` statement

The `case` Statement

```
$ cat case_demo
#!/bin/sh
echo "  Use one of the following options:"
echo "  d or D:   To display today's date and present time"
echo "  l or L:   To see the listing of files in your present working directory"
echo "  w or W:   To see who's logged in"
echo "  q or Q:   To quit this program"
echo "Enter your option and hit <Enter>: \c"
read option
case "$option" in
       d|D)     date
                ;;
       l|L)   ls
                ;;
       w|W)   who
                ;;
       q|Q)   exit 0
                ;;
       *)     echo "Invalid option; try running the program again."
                exit 1
                  ;;
esac
exit 0
$
```

The `case` Statement

```
$ case_demo
Use one of the following options:
       d or D:  To display today's date and present time
       l or L:  To see the listing of files in your present working directory
       w or W:  To see who is logged in
       q or Q:  To quit this program
Enter your option and hit <Enter>: D
Sat June 12 18:14:22 PDT 2004
$ case_demo
Use one of the following options:
       d or D:  To display today's date and present time
       l or L:  To see the listing of files in your present working directory
       w or W:  To see who is logged in
       q or Q:  To quit this program
Enter your option and hit <Enter>: a
Invalid option; try running the program again.
$
```