2.1 Taking a Peek at Shell Scripts

If you read, write, or maintain programs, the following samples will give you a quick overview of the construction and style of a shell script and introduce you to some of the constructs and syntax found in these programs. *Note: If you are not familiar with programming, skip this chapter and go to Chapter 3.* When you have finished learning how to write scripts, you may want to return to this chapter for a quick reference to refresh your memory.

The C shell and TC shell emulate the C language syntax whereas the Bourne shell is based on an older programming language called Algol.

The Bash and Korn shells tend to be a combination of both the Bourne and C shells, although these shells originated from the Bourne shell.

To illustrate the differences in the shells, four sample programs are provided, one for each shell (the C and TC shells are presented together here). Above each program, a list of basic constructs are described for the shell being examined.

2.2 Sample Scripts: Comparing the Major Shells

At the end of each section pertaining to a specific shell, you will find a small program to illustrate how to write a complete script. At first glance, the programs for each shell look very similar. They are. And they all do the same thing. The main difference is the syntax. After you have worked with these shells for some time, you will quickly adapt to the differences and start formulating your own opinions about which shell is your favorite. A detailed comparison of differences among the C/TC, Bourne, Bash, and Korn shells is found in Appendix B.
Before Getting Started. You must have a good handle on UNIX/Linux commands. If you do not know the basic commands, you cannot do much with shell programming. The next three chapters will teach you how to use some of the major UNIX/Linux commands, and Appendix A in the back of the book, gives you a list of the most common commands (also called utilities).

The Purpose. The sample scripts provided at the end of each section send a mail message to a list of users, inviting each of them to a party. The place and time of the party are set in variables. The list of guests is selected from a file called guests. The existence of the guest file is checked and if it does not exist, the program will exit. A list of foods is stored in a word list (array). A loop is used to iterate through the list of guests. Each user will receive an e-mail invitation telling him or her the time and place of the party and asking him or her to bring an item from the food list. A conditional is used to check for a user named root, and if he is on the guest list, he will be excluded; that is, he will not be sent an e-mail invitation. The loop will continue until the guest list is empty. Each time through the loop, a food item is removed from the list, so that each guest will be asked to bring a different food. If, however, there are more users than foods, the list is reset. This is handled with a standard loop control statement.

2.3 The C and TC Shell Syntax and Constructs

The basic C and TC shell syntax and constructs are listed in Table 2.1.

<table>
<thead>
<tr>
<th>Table 2.1</th>
<th>C and TC Shell Syntax and Constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>The shbang line</td>
<td>The “shbang” line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a hash mark #, an exclamation point ! (called a bang), followed by the full pathname of the shell, and any shell options. Any other lines beginning with a # are used as comments.</td>
</tr>
<tr>
<td>Example</td>
<td>#!/bin/csh or #!/bin/tcsh</td>
</tr>
<tr>
<td>Comments</td>
<td>Comments are descriptive material preceded by a # sign; they are not executable statements. They are in effect until the end of a line and can be started anywhere on the line.</td>
</tr>
<tr>
<td>Example</td>
<td># This is a comment</td>
</tr>
</tbody>
</table>
### 2.3 The C and TC Shell Syntax and Constructs

#### Table 2.1 C and TC Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wildcards</td>
<td>There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or “wildcards.” These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The ! is the history character, the &lt;, &gt;, &gt;&gt;, &lt;&amp;, and</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>rm *; ls ??; cat file[1-3]; !! echo &quot;How are you?&quot; echo Oh boy!</td>
</tr>
<tr>
<td>Displaying output</td>
<td>To print output to the screen, the echo command is used. Wildcards must be escaped with either a backslash or matching quotes.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>echo &quot;Hello to you!&quot;</td>
</tr>
<tr>
<td>Local variables</td>
<td>Local variables are in scope for the current shell. When a script ends or the shell exits, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>set variable_name = value set name = &quot;Tom Jones&quot;</td>
</tr>
<tr>
<td>Global variables</td>
<td>Global variables are called environment variables. They are set for the currently running shell and are available to any process spawned from that shell. They go out of scope when the script ends or the shell where they are defined exits.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>setenv VARIABLE_NAME value setenv PRINTER Shakespeare</td>
</tr>
<tr>
<td>Extracting values from variables</td>
<td>To extract the value from variables, a dollar sign is used.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>echo $variable_name echo $name echo $PRINTER</td>
</tr>
<tr>
<td>Reading user input</td>
<td>The special variable $&lt; reads a line of input from the user and assigns it to a variable.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>echo &quot;What is your name?&quot; set name = $&lt;</td>
</tr>
</tbody>
</table>
Arguments can be passed to a script from the command line. Two methods can be used to receive their values from within the script: positional parameters and the `argv` array.

**EXAMPLE**

% `scriptname arg1 arg2 arg3 ...`

Using positional parameters:

```bash
echo $1 $2 $3
echo $*
```

Using the `argv` array:

```bash
echo $argv[*] $argv
```

Arrays

An array is a list of words separated by whitespace. The list is enclosed in a set of parentheses.

The built-in `shift` command shifts off the left-hand word in the list.

Unlike C, the individual words are accessed by index values, which start at 1 rather than 0.

**EXAMPLE**

```bash
set word_list = ( word1 word2 word3 )
set names = ( Tom Dick Harry Fred )
shift names
```

```bash
echo $word_list[1] displays first element of the list
echo $word_list[2] displays second element of the list
echo $word_list[3] displays all elements of the list
echo $names[1]
echo $names[2]
echo $names[3]
echo $names[*]
```

Command substitution

To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, the command is enclosed in backquotes.

**EXAMPLE**

```bash
set variable_name=`command`
set now = `date`
```

```bash
echo $variable_name
echo "Today is `date`"
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Arguments can be passed to a script from the command line. Two methods can be used to receive their values from within the script: positional parameters and the <code>argv</code> array.</td>
</tr>
<tr>
<td>Arrays</td>
<td>An array is a list of words separated by whitespace. The list is enclosed in a set of parentheses. The built-in <code>shift</code> command shifts off the left-hand word in the list. Unlike C, the individual words are accessed by index values, which start at 1 rather than 0.</td>
</tr>
<tr>
<td>Command substitution</td>
<td>To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, the command is enclosed in backquotes.</td>
</tr>
</tbody>
</table>
2.3 The C and TC Shell Syntax and Constructs

Table 2.1 C and TC Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Variables that will hold the results of an arithmetic computation must be preceded by an <code>@</code> symbol and a space. Only integer arithmetic is provided by this shell.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE</strong></td>
<td><code>@ n = 5 + 5</code>&lt;br&gt;<code>echo $n</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operators</th>
<th>The C and TC shells support operators for testing strings and numbers similar to those found in the C language.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>Equality: <code>&lt;</code> greater than <code>&gt;</code> greater than or equal to <code>&lt;=</code> less than or equal to <code>&gt;=</code> greater than or equal to&lt;br&gt;Logical: <code>&amp;&amp;</code> and `</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditional statements</th>
<th>The if construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The <code>then</code> keyword is placed after the closing parentheses. An if must end with an <code>endif</code>. An alternative to <code>if/else if</code> is the <code>switch</code> statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE</strong></td>
<td>The if construct is:&lt;br&gt;<code>if ( expression ) then</code>&lt;br&gt;block of statements&lt;br&gt;<code>endif</code> The if/else/else if construct is:&lt;br&gt;<code>if ( expression ) then</code>&lt;br&gt;block of statements&lt;br&gt;<code>else if ( expression ) then</code>&lt;br&gt;block of statements&lt;br&gt;<code>else if ( expression ) then</code>&lt;br&gt;block of statements&lt;br&gt;<code>else</code>&lt;br&gt;block of statements&lt;br&gt;<code>endif</code></td>
</tr>
</tbody>
</table>

| | | |
Conditional statements (continued)

The switch construct is:

```bash
switch variable_name
  case constant1:
    statements
  case constant2:
    statements
  case constant3:
    statements
  default:
    statements
endsw
```

```bash
switch ( "$color" )
  case blue:
    echo $color is blue
    breaksw
  case green:
    echo $color is green
    breaksw
  case red:
    case orange:
      echo $color is red or orange
      breaksw
    default:
      echo "Not a valid color"
endsw
```

Loops

There are two types of loops; the while and foreach loop.

The while loop is followed by an expression enclosed in parentheses, a block of statements, and terminated with the end keyword. As long as the expression is true, the looping continues.

The foreach loop is followed by a variable name and a list of words enclosed in parentheses, a block of statements, and terminates with the end keyword. The foreach loop iterates through a list of words, processing a word and then shifting it off, then moving to the next word. When all words have been shifted from the list, it ends.

The loop control commands are break and continue.

```bash
while ( expression )
  block of statements
end

foreach variable ( word list )
  block of statements
end

-------------------------------
foreach color (red green blue)
  echo $color
end
```

File testing

The C shell has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth. For other types of file tests, the UNIX test command is used. See Example 2.1 for a demonstration.

```bash
Table 2.1  C and TC Shell Syntax and Constructs (continued)

| Conditional statements (continued) | switch construct:
|-----------------------------------|--------------------------------------------------|
| switch variable_name:             | switch ( "$color" ):
| case constant1:                   |   case blue:
|    statements                      |     echo $color is blue
| case constant2:                   |     breaksw
|    statements                      | case green:
| case constant3:                   |     echo $color is green
|    statements                      |     breaksw
| default:                          | case red:
|    statements                      | case orange:
|                                     |     echo $color is red or orange
|                                     |     breaksw
|                                     | default:
|                                     |     echo "Not a valid color"
|                                     | endsw
```
2.3 The C and TC Shell Syntax and Constructs

Table 2.1 C and TC Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>File testing (continued)</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>–r</td>
<td>Current user can read the file</td>
</tr>
<tr>
<td>–w</td>
<td>Current user can write to the file</td>
</tr>
<tr>
<td>–x</td>
<td>Current user can execute the file</td>
</tr>
<tr>
<td>–e</td>
<td>File exists</td>
</tr>
<tr>
<td>–o</td>
<td>Current user owns the file</td>
</tr>
<tr>
<td>–z</td>
<td>File is zero length</td>
</tr>
<tr>
<td>–d</td>
<td>File is a directory</td>
</tr>
<tr>
<td>–f</td>
<td>File is a plain file</td>
</tr>
</tbody>
</table>

EXAMPLE 2.1

```csh
#!/bin/csh -f
1 if ( –e file ) then
   echo file exists
endif

2 if ( –d file ) then
   echo file is a directory
endif

3 if ( ! –z file ) then
   echo file is not of zero length
endif

4 if ( –r file && –w file ) then
   echo file is readable and writable
endif
```

2.3.1 The C/TC Shell Script

The program in Example 2.2 is an example of a C shell/TC shell script. The program contains many of the constructs discussed in Table 2.1.

EXAMPLE 2.2

```csh
#!/bin/csh -f
# The Party Program--Invitations to friends from the "guest" file
3 set guestfile = ~/shell/guests
4 if ( ! –e "$guestfile" ) then
   echo "$guestfile:t non-existent"
   exit 1
endif
5 setenv PLACE "Sarotini's"
```
EXAMPLE 2.2 (CONTINUED)

```bash
7 \@ Time = 'date +%H' + 1
8 set food = ( cheese crackers shrimp drinks "hot dogs" sandwiches )
9 foreach person ( "cat $guestfile" )
10 if ( $person =~ root ) continue
11 mail –v –s "Party" $person << FINIS   # Start of here document
12   Hi $person! Please join me at $PLACE for a party!
13   Meet me at $Time o'clock.
14   I'll bring the ice cream. Would you please bring $food[1] and
15   anything else you would like to eat? Let me know if you can
16   make it. Hope to see you soon.
17     Your pal,
18     ellie@`hostname`       # or `uname -n`
19 FINIS
20 shift food
21 if ( $#food == 0 ) then
22   set food = ( cheese crackers shrimp drinks "hot dogs"
23               sandwiches )
24 endif
25 end
26 echo "Bye..."
```

EXPLANATION

1 This line lets the kernel know that you are running a C shell script. The -f option is a fast startup. It says, “Do not execute the .cshrc file,” an initialization file that is automatically executed every time a new csh program is started.

2 This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.

3 The variable guestfile is set to the full pathname of a file called guests.

4 This line reads: If the file guests does not exist, then print to the screen “guests nonexistent” and exit from the script with an exit status of 1 to indicate that something went wrong in the program.

5 This marks the end of the statements based on the if condition.

6 Variables are assigned the values for the place and time. The PLACE variable is an environment variable.

7 The Time variable is a local variable. The @ symbol tells the C shell to perform its built-in arithmetic; that is, add 1 to the Time variable after extracting the hour from the date command. The Time variable is spelled with an uppercase T to prevent the C shell from confusing it with one of its reserved words, time.

8 The food array is created. It consists of a list of words separated by whitespace. Each word is an element of the food array.
2.4 The Bourne Shell Syntax and Constructs

The basic Bourne shell syntax and constructs are listed in Table 2.2.

<table>
<thead>
<tr>
<th>The shbang line</th>
<th>The “shbang” line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a <code>#!</code> followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE</strong></td>
<td><code>#!/bin/sh</code></td>
</tr>
<tr>
<td>Comments</td>
<td>Comments are descriptive material preceded by a <code>#</code> sign. They are in effect until the end of a line and can be started anywhere on the line.</td>
</tr>
</tbody>
</table>
| **EXAMPLE**     | `# this text is not`  
|                 | `# interpreted by the shell`                                                                                         |
### Wildcards

There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

**Example**

Filename expansion:
```bash
rm *; ls ??; cat file[1-3];
```

Quotes protect metacharacter:
```bash
echo "How are you?"
```

### Displaying output

To print output to the screen, the `echo` command is used. Wildcards must be escaped with either a backslash or matching quotes.

**Example**

```bash
echo "What is your name?"
```

### Local variables

Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

**Example**

```bash
variable_name=value
name="John Doe"
x=5
```

### Global variables

Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

**Example**

```bash
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:. export PATH
```

### Extracting values from variables

To extract the value from variables, a dollar sign is used.

**Example**

```bash
echo $variable_name
echo $name
echo $PATH
```
### 2.4 The Bourne Shell Syntax and Constructs

**Table 2.2 Bourne Shell Syntax and Constructs (continued)**

<table>
<thead>
<tr>
<th>Reading user input</th>
<th>The <code>read</code> command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The <code>read</code> command can accept multiple variable names. Each variable will be assigned a word.</th>
</tr>
</thead>
</table>
| **Example**         | `echo "What is your name?"
read name
read name1 name2 ...` |

<table>
<thead>
<tr>
<th>Arguments (positional parameters)</th>
<th>Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.</th>
</tr>
</thead>
</table>
| **Example**                       | At the command line: $ scriptname arg1 arg2 arg3 ...

- `echo $1 $2 $3` *Positional parameters*
- `echo $*` *All the positional parameters*
- `echo $#` *The number of positional parameters*

<table>
<thead>
<tr>
<th>Arrays (positional parameters)</th>
<th>The Bourne shell does support an array, but a word list can be created by using positional parameters. A list of words follows the built-in <code>set</code> command, and the words are accessed by position. Up to nine positions are allowed. The built-in <code>shift</code> command shifts off the first word on the left-hand side of the list. The individual words are accessed by position values starting at 1.</th>
</tr>
</thead>
</table>
| **Example**                     | `set word1 word2 word3
echo $1 $2 $3` *Displays word1, word2, and word3*

- `set apples peaches plums
shift
echo $1` *Shifts off apples*
- `echo $2` *Displays first element of the list*
- `echo $*` *Displays all elements of the list*

<table>
<thead>
<tr>
<th>Command substitution</th>
<th>To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, backquotes are used.</th>
</tr>
</thead>
</table>
| **Example**          | `variable_name='command'
echo $variable_name

now='date'
echo $now
echo "Today is 'date'"` |
### Chapter 2 • Shell Programming QuickStart

#### Table 2.2 Bourne Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>The Bourne shell does not support arithmetic. UNIX/Linux commands must be used to perform calculations.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE</strong></td>
<td></td>
</tr>
<tr>
<td>n=<code>expr 5 + 5</code></td>
<td></td>
</tr>
<tr>
<td>echo $n</td>
<td></td>
</tr>
<tr>
<td>Operators</td>
<td>The Bourne shell uses the built-in test command operators to test numbers and strings.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td></td>
</tr>
<tr>
<td>Equality:</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>string</td>
</tr>
<tr>
<td>!=</td>
<td>string</td>
</tr>
<tr>
<td>-eq</td>
<td>number</td>
</tr>
<tr>
<td>-ne</td>
<td>number</td>
</tr>
<tr>
<td>Logical:</td>
<td></td>
</tr>
<tr>
<td>-a</td>
<td>and</td>
</tr>
<tr>
<td>-o</td>
<td>or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>Relational:</td>
<td></td>
</tr>
<tr>
<td>-gt</td>
<td>greater than</td>
</tr>
<tr>
<td>-ge</td>
<td>greater than, equal to</td>
</tr>
<tr>
<td>-lt</td>
<td>less than</td>
</tr>
<tr>
<td>-le</td>
<td>less than, equal to</td>
</tr>
<tr>
<td>Conditional statements</td>
<td>The if construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The then keyword is placed after the closing parenthesis. An if must end with a fi.</td>
</tr>
<tr>
<td><strong>EXAMPLE</strong></td>
<td></td>
</tr>
<tr>
<td>The if construct is:</td>
<td></td>
</tr>
<tr>
<td>if command</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>block of statements</td>
<td></td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>The if/else construct is:</td>
<td></td>
</tr>
<tr>
<td>if [ expression ]</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>block of statements</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>block of statements</td>
<td></td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>if [ expression ]</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>block of statements</td>
<td></td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
</tbody>
</table>
2.4 The Bourne Shell Syntax and Constructs

Table 2.2 Bourne Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Conditional statements (continued)</th>
<th>The if/else/elif construct is:</th>
<th>The case command construct is:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>if command then block of statements elif command then block of statements elif command then block of statements else block of statements fi</td>
<td>case variable_name in pattern1) statements pattern2) statements pattern3) else *) default value fi</td>
</tr>
<tr>
<td></td>
<td>-----------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td></td>
<td>if [ expression ] then block of statements elseif [ expression ] then block of statements elseif [ expression ] then block of statements else block of statements fi</td>
<td>case &quot;$color&quot; in blue) block of statements green) block of statements red(orange) block of statements *) echo &quot;Not a color&quot; # default esac fi</td>
</tr>
</tbody>
</table>

Loops

There are three types of loops: while, until and for.

The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.

The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.

The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.

The loop control commands are break and continue.

**EXAMPLE**

```bash
while command
do    block of statementsdone
while [ expression ]
do    block of statementsdone
```
Chapter 2 • Shell Programming QuickStart

Table 2.2 Bourne Shell Syntax and Constructs (continued)

| Loops (continued) | until command do block of statements done until [ expression ] do block of statements done for variable word1 word2 word3 ... do block of statements done |

File testing The Bourne shell uses the test command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth. See Example 2.3.

**EXAMPLE**

- d File is a directory
- f File exists and is not a directory
- r Current user can read the file
- s File is of nonzero size
- w Current user can write to the file
- x Current user can execute the file

**EXAMPLE 2.3**

```bash
#!/bin/sh
1 if [-f file ]
   then
      echo file exists
   fi
2 if [-d file ]
   then
      echo file is a directory
   fi
3 if [-s file ]
   then
      echo file is not of zero length
   fi
4 if [-r file -a -w file ]
   then
      echo file is readable and writable
   fi
```
## 2.4 The Bourne Shell Syntax and Constructs

### Table 2.2 Bourne Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Functions</th>
<th>Functions allow you to define a section of shell code and give it a name. The Bourne shell introduced the concept of functions. The C and TC shells do not have functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE</td>
<td>function_name() { block of code } [-----------------------] [-----------------------] lister() { echo Your present working directory is <code>pwd</code> echo Your files are: ls }</td>
</tr>
</tbody>
</table>

### 2.4.1 The Bourne Shell Script

### EXAMPLE 2.4

```bash
#!/bin/sh
# The Party Program--Invitations to friends from the "guest" file
guestfile=/home/jody/ellie/shell/guests
if [ ! -f "$guestfile" ]; then
  echo "basename $guestfile" non-existent
  exit 1
fi
PLACE="Sarotini's"; export PLACE
Time=`date +%H`
Time=`expr $Time + 1`
set cheese crackers shrimp drinks "hot dogs" sandwiches
for person in "cat $guestfile"
do
  if [ $person =~ root ]; then
    continue
  else
    # mail -v -s "Party" $person <<< FINIS
```

Table 2.2 Bourne Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Functions</th>
<th>Functions allow you to define a section of shell code and give it a name. The Bourne shell introduced the concept of functions. The C and TC shells do not have functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE</td>
<td>function_name() { block of code } [-----------------------] [-----------------------] lister() { echo Your present working directory is <code>pwd</code> echo Your files are: ls }</td>
</tr>
</tbody>
</table>

### 2.4.1 The Bourne Shell Script

### EXAMPLE 2.4

```bash
#!/bin/sh
# The Party Program--Invitations to friends from the "guest" file
guestfile=/home/jody/ellie/shell/guests
if [ ! -f "$guestfile" ]; then
  echo "basename $guestfile" non-existent
  exit 1
fi
PLACE="Sarotini's"; export PLACE
Time=`date +%H`
Time=`expr $Time + 1`
set cheese crackers shrimp drinks "hot dogs" sandwiches
for person in "cat $guestfile"
do
  if [ $person =~ root ]; then
    continue
  else
    # mail -v -s "Party" $person <<< FINIS
```
17 cat <<FINIS
Hi ${person}! Please join me at $PLACE for a party!
Meet me at $Time o'clock.
I'll bring the ice cream. Would you please bring $1 and
anything else you would like to eat? Let me know if you
can make it. Hope to see you soon.
Your pal,
    ellie@`hostname`
FINIS
18 shift
19 if [ $# -eq 0 ]
then
    set cheese crackers shrimp drinks "hot dogs" sandwiches
fi
21 done
echo "Bye..."

EXPLANATION
1 This line lets the kernel know that you are running a Bourne shell script.
2 This is a comment. It is ignored by the shell, but important for anyone trying to
   understand what the script is doing.
3 The variable guestfile is set to the full pathname of a file called guests.
4 This line reads: If the file guests does not exist, then print to the screen "guests non-
   existent" and exit from the script.
5 The then is usually on a line by itself, or on the same line as the if statement if it
   is preceded by a semicolon.
6 The UNIX basename command removes all but the filename in a search path. Be-
   cause the command is enclosed in backquotes, command substitution will be per-
   formed and the output displayed by the echo command.
7 If the file does not exist, the program will exit. An exit with a value of 1 indicates
   that there was a failure in the program.
8 The fi keyword marks the end of the block of if statements.
9 Variables are assigned the values for the place and time. PLACE is an environment
   variable, because after it is set, it is exported.
10 The value in the Time variable is the result of command substitution; i.e., the out-
    put of the date +%H command (the current hour) will be assigned to Time.
11 The list of foods to bring is assigned to special variables (positional parameters)
   with the set command.
12 The for loop is entered. It loops through until each person listed in the guest file
   has been processed.
2.5 The Korn Shell Constructs

The Korn and Bash shells are very similar. The following constructs will work for both shells. To see all the subtle variations, see the individual chapters for these shells.

Table 2.3 Korn Shell Syntax and Constructs

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>The shbang line</td>
<td><code>#!/bin/ksh</code></td>
</tr>
<tr>
<td>Comments</td>
<td><code># This program will test some files</code></td>
</tr>
</tbody>
</table>

EXPLANATION (CONTINUED)

13 If the variable `person` matches the name of the user `root`, loop control will go to the top of the `for` loop and process the next person on the list. The user `root` will not get an invitation.

14 The `continue` statement causes loop control to start at line 12, rather than continuing to line 16.

15 The block of statements under `else` are executed if line 13 is not true.

16 The mail message is sent when this line is uncommented. It is a good idea to comment this line until the program has been thoroughly debugged, otherwise the e-mail will be sent to the same people every time the script is tested.

17 The next statement, using the `cat` command with the `here document`, allows the script to be tested by sending output to the screen that would normally be sent through the mail when line 7 is uncommented.

18 After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, ensuring that each person is assigned a food.

19 The value of `$#` is the number of positional parameters left. If that number is 0, the food list is empty.

20 The food list is reset.

21 The `done` keyword marks the end of the block of statements in the body of the `for` loop.
Wildcards | There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

**Example**

```
rm *; ls ??; cat file[1-3];
echo "How are you?"
```

Displaying output | To print output to the screen, the `echo` command can be used. Wildcards must be escaped with either a backslash or matching quotes. Korn shell also provides a built-in `print` function to replace the `echo` command.

**Example**

```
echo "Who are you?"
print "How are you?"
```

Local variables | Local variables are in scope for the current shell. When a script ends or the shell exits, they are no longer available; i.e., they go out of scope. The `typeset` built-in command can also be used to declare variables. Local variables are set and assigned values.

**Example**

```
variable_name=value
`typeset` variable_name=value
name="John Doe"
x=5
```

Global variables | Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends or the shell where they were defined exits.

**Example**

```
export VARIABLE_NAME =value
export PATH=/bin:/usr/bin:
```

Extracting values from variables | To extract the value from variables, a dollar sign is used.

**Example**

```
echo $variable_name
echo $name
echo $PATH
```

Reading user input | The user will be asked to enter input. The `read` command is used to accept a line of input. Multiple arguments to `read` will cause a line to be broken into words, and each word will be assigned to the named variable. The Korn shell allows the prompt and `read` command to be combined.
### 2.5 The Korn Shell Constructs

#### Table 2.3 Korn Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Reading user input (continued)</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read name?&quot;What is your name?&quot;</code></td>
<td>The prompt is in quotes. After it is displayed, the read command waits for user input.</td>
</tr>
</tbody>
</table>
| `print -n "What is your name?"
read name`
| `read name1 name2 ...` |

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

At the command line:

```
$ scriptname arg1 arg2 arg3 ...
```

In a script:

```
echo $1 $2 $3           Positional parameters, $1 is assigned arg1, $2 is assigned arg2, ...
echo $*                  All the positional parameters
echo $#                  The number of positional parameters
```

<table>
<thead>
<tr>
<th>Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Bourne shell utilizes positional parameters to create a word list. In addition to positional parameters, the Korn shell also supports an array syntax whereby the elements are accessed with a subscript, starting at 0. Korn shell arrays are created with the <code>set -A</code> command.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

```
set apples pears peaches
print $1 $2 $3  Positional parameters
set -A array_name word1 word2 word3 ...  Array
print ${fruit[0]}  Prints apples
${fruit[1]} = oranges  Assign a new value
```

<table>
<thead>
<tr>
<th>Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Korn shell supports integer arithmetic. The typeset -i command will declare an integer type variable. Integer arithmetic can be performed on variables declared this way. Otherwise, the <code>(( ))</code> syntax (let command) is used for arithmetic operations.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

```
typeset -i variable_name  Declare integer
typeset -i num            num is declared as an integer
num=5+4
print $num                 Prints 9
(( n=5 + 5 ))              The let command
print $n                  Prints 20
```
Like the C/TC shells and the Bourne shell, the output of a UNIX/Linux command can be assigned to a variable, or used as the output of a command in a string, by enclosing the command in backquotes. The Korn shell also provides a new syntax. Instead of placing the command between backquotes, it is enclosed in a set of parentheses, preceded by a dollar sign.

**Example**

```bash
variable_name=`command`
variable_name=$( command )
echo $variable_name
echo "Today is `date`"
echo "Today is $(date)"
```

**Operators**

The Korn shell uses the built-in test command operators to test numbers and strings, similar to C language operators.

**Example**

- **Equality:**
  - `=`  string, equal to
  - `!=` string, not equal to
  - `==` number, equal to
  - `!=` number, not equal to

- **Logical:**
  - `&&`  and
  - `||`  or
  - `!`  not

- **Relational:**
  - `>`  greater than
  - `=>` greater than, equal to
  - `<`  less than
  - `<=` less than, equal to

**Conditional statements**

The `if` construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The `then` keyword is placed after the closing parenthesis. An `if` must end with a `fi`. The new test command `[[ ]]` is now used to allow pattern matching in conditional expressions. The old test command `[ ]` is still available for backward compatibility with the Bourne shell. The `case` command is an alternative to `if/else`.

**Example**

The `if` construct is:

```bash
if command
  then
    block of statements
fi
```

```bash
if [[ numeric expression ]]
  then
    block of statements
fi
```

```bash
if [[ string expression ]]
  then
    block of statements
fi
```
2.5 The Korn Shell Constructs

Table 2.3 Korn Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Conditional statements (continued)</th>
<th>The if/else construct is:</th>
<th>The if/else/else if construct is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if command then block of statements else block of statements fi</td>
<td>if [[ expression ]] then block of statements else block of statements fi</td>
<td>if [[ string expression ]] then block of statements elif [[ string expression ]] then block of statements elif [[ string expression ]] then block of statements else block of statements fi</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>if (( numeric expression )) then block of statements else block of statements fi</td>
<td>if (( numeric expression )) then block of statements elif (( numeric expression )) then block of statements elif (( numeric expression )) then block of statements else block of statements fi</td>
<td>if (( numeric expression )) then block of statements elif (( numeric expression )) then block of statements elif (( numeric expression )) then block of statements else block of statements fi</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>The case construct is: case variable_name in pattern1 statements ;; pattern2 statements ;; pattern3 ;; esac</td>
<td>case &quot;$color&quot; in blue) echo $color is blue ;; green) echo $color is green ;; red/orange) echo $color is red or orange ;; esac</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2  •  Shell Programming QuickStart

Table 2.3  Korn Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Loops</th>
<th>There are four types of loops: while, until, for, and select.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The while loop is followed by an expression enclosed in square</td>
</tr>
<tr>
<td></td>
<td>brackets, a do keyword, a block of statements, and terminated</td>
</tr>
<tr>
<td></td>
<td>with the done keyword. As long as the expression is true, the</td>
</tr>
<tr>
<td></td>
<td>body of statements between do and done will be executed.</td>
</tr>
<tr>
<td></td>
<td>The until loop is just like the while loop, except the body of</td>
</tr>
<tr>
<td></td>
<td>the loop will be executed as long as the expression is false.</td>
</tr>
<tr>
<td></td>
<td>The for loop is used to iterate through a list of words,</td>
</tr>
<tr>
<td></td>
<td>processing a word and then shifting it off, to process the next</td>
</tr>
<tr>
<td></td>
<td>word. When all words have been shifted from the list, it ends.</td>
</tr>
<tr>
<td></td>
<td>The select loop is used to provide a prompt (PS3 variable) and</td>
</tr>
<tr>
<td></td>
<td>a menu of numbered items from which the user inputs a selection</td>
</tr>
<tr>
<td></td>
<td>The input will be stored in the special built-in REPLY variable.</td>
</tr>
<tr>
<td></td>
<td>The select loop is normally used with the case command.</td>
</tr>
<tr>
<td></td>
<td>The loop control commands are break and continue. The break</td>
</tr>
<tr>
<td></td>
<td>command allows control to exit the loop before reaching the end</td>
</tr>
<tr>
<td></td>
<td>of it; the continue command allows control to return to the</td>
</tr>
<tr>
<td></td>
<td>looping expression before reaching the end.</td>
</tr>
</tbody>
</table>

**EXAMPLE**

```bash
while command
do
    block of statements
done
---------
while [[ string expression ]]
do
    block of statements
done
---------
while (( numeric expression ))
do
    block of statements
done
---------
until command
do
    block of statements
done
---------
until [[ string expression ]]
do
    block of statements
done
---------
until (( numeric expression ))
do
    block of statements
done

select variable in word_list
do
    block of statements
done
---------
for variable in word_list
do
    block of statements
done
---------
for name in Tom Dick Harry
do
    print "Hi $name"
done
---------
for item in blue red green
echo $item
done
---------
PS3="Select an item from the menu"
for item in blue red green
echo $item
done
---------
Shows menu:
    1) blue
    2) red
    3) green
```
2.5 The Korn Shell Constructs

File testing

The Korn shell uses the `test` command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth. See Example 2.5.

**EXAMPLE**

```sh
#!/bin/sh
1 if [ -a file ]
   then
    echo file exists
   fi
2 if [ -d file ]
   then
    echo file is a directory
   fi
3 if [ -s file ]
   then
    echo file is not of zero length
   fi
4 if [ -r file -a -w file ]
   then
    echo file is readable and writable
   fi
```

Functions

Functions allow you to define a section of shell code and give it a name. There are two formats: one from the Bourne shell, and the Korn shell version that uses the `function` keyword.

**EXAMPLE**

```sh
function_name() {
   block of code
}
function function_name {
   block of code
}-------------------------
```
2.5.1 The Korn Shell Script

Example 2.6

```
#!/bin/ksh
# The Party Program—Invitations to friends from the "guest" file
guestfile=~/shell/guests
if [[ ! -a "$guestfile" ]]
    then
        print "${guestfile##*/} non-existent"
        exit 1
    fi
export PLACE="Sarotini's"
(( Time=$(date +%H) + 1 ))
set -A foods cheese crackers shrimp drinks "hot dogs" sandwiches
typeset -i n=0
for person in $(< $guestfile)
do
    if [[ $person = root ]]
        then
            continue
        else
            # Start of here document
            mail –v –s "Party" $person <<- FINIS
            Hi ${person}! Please join me at $PLACE for a party!
            Meet me at $Time o'clock.
            I'll bring the ice cream. Would you please bring
            ${foods[n]} and anything else you would like to eat? Let
            me know if you can make it.
            Hope to see you soon.
            Your pal,
            ellie@`hostname`
            FINIS
            n=n+1
        fi
```

Table 2.3 Korn Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Functions (continued)</th>
<th>Function</th>
<th>lister</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>echo Your present working directory is <code>pwd</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>echo Your files are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ls</td>
<td></td>
</tr>
</tbody>
</table>

EXAMPLE

2.6 (CONTINUED)

```bash
if (( ${#foods[*]} == $n ))
  then
    set -A foods cheese crackers shrimp drinks "hot dogs" sandwiches
  fi
fi
done
print "Bye..."
```

EXPLANATION

1. This line lets the kernel know that you are running a Korn shell script.
2. This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.
3. The variable guestfile is set to the full pathname of a file called guests.
4. This line reads: If the file guests does not exist, then print to the screen “guests non-existent” and exit from the script.
5. An environment variable is assigned a value and exported (made available to sub-shells).
6. The output of the UNIX/Linux command, the hour of the day, is assigned to the variable called Time. Variables are assigned the values for the place and time.
7. The list of foods to bring is assigned to an array called foods with the set -A command. Each item on the list can be accessed with an index starting at 0.
8. The typeset -i command is used to create an integer value.
9. For each person on the guest list a mail message will be created inviting the person to a party at a given place and time, and assigning a food from the list to bring.
10. The condition tests for the user root. If the user is root, control will go back to the top of the loop and assign the next user in the guest list to the variable, person.
11. The mail message is sent. The message body is contained in a here document.
12. The variable n is incremented by 1.
13. If the number of elements in the array is equal to the value of the variable, then the end of the array has been reached.
14. This marks the end of the looping statements.
## 2.6 The Bash Shell Constructs

The Korn and Bash shells are very similar, but there are some differences. The Bash constructs are listed in Table 2.4.

<table>
<thead>
<tr>
<th>The shbang line</th>
<th>The “shbang” line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a <code>#!</code> followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td><code>#!/bin/bash</code></td>
</tr>
<tr>
<td>Comments</td>
<td>Comments are descriptive material preceded by a <code>#</code> sign. They are in effect until the end of a line and can be started anywhere on the line.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><code># This is a comment</code></td>
</tr>
<tr>
<td>Wildcards</td>
<td>There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or “wildcards.” These characters are neither numbers or letters. For example, the <code>*</code>, <code>?</code>, and <code>[ ]</code> are used for filename expansion. The <code>&lt;</code>, <code>&gt;</code>, <code>2&gt;</code>, <code>&gt;&gt;</code>, and `</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><code>rm *; ls ??; cat file[1-3]; echo &quot;How are you?&quot;</code></td>
</tr>
<tr>
<td>Displaying output</td>
<td>To print output to the screen, the <code>echo</code> command is used. Wildcards must be escaped with either a backslash or matching quotes.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><code>echo &quot;How are you?&quot;</code></td>
</tr>
<tr>
<td>Local variables</td>
<td>Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables can also be defined with the built-in <code>declare</code> function. Local variables are set and assigned values.</td>
</tr>
</tbody>
</table>
| **Example**    | `variable_name=value
declare variable_name=value
name="John Doe"
x=5`                                                                                                                                  |
Global variables are called environment variables and are created with the `export` built-in command. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

The built-in `declare` function with the `-x` option also sets an environment variable and marks it for export.

```
export VARIABLE_NAME=value
declare -x VARIABLE_NAME=value
export PATH=/bin:/usr/bin:.  
```

To extract the value from variables, a dollar sign is used.

```
echo $variable_name
echo $name
echo $PATH
```

The user will be asked to enter input. The `read` command is used to accept a line of input. Multiple arguments to `read` will cause a line to be broken into words, and each word will be assigned to the named variable.

```
echo "What is your name?"
read name
read name1 name2 ...
```

Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.

```
At the command line:
$ scriptname arg1 arg2 arg3 ...

In a script:
echo $1 $2 $3          Positional parameters
echo $*                All the positional parameters
echo $#                The number of positional parameters
```

The Bourne shell utilizes positional parameters to create a word list. In addition to positional parameters, the Bash shell supports an array syntax whereby the elements are accessed with a subscript, starting at 0. Bash shell arrays are created with the `declare -a` command.
Arrays (continued)

<table>
<thead>
<tr>
<th>EXAMPLE</th>
</tr>
</thead>
</table>
| set apples pears peaches  (positional parameters)  
echo $1 $2 $3 |
| declare -a array_name=(word1 word2 word3 ...)
declare -a fruit=(apples pears plums)
echo ${fruit[0]} |

Command substitution

Like the C/TC shells and the Bourne shell, the output of a UNIX/Linux command can be assigned to a variable, or used as the output of a command in a string, by enclosing the command in backquotes. The Bash shell also provides a new syntax. Instead of placing the command between backquotes, it is enclosed in a set of parentheses, preceded by a dollar sign.

EXAMPLE

variable_name=`command`
variable_name=$( command )
echo $variable_name

echo "Today is `date"
echo "Today is $(date)"

Arithmetic

The Bash shells support integer arithmetic. The declare -i command will declare an integer type variable. The Korn shell's typeset command can also be used for backward compatibility. Integer arithmetic can be performed on variables declared this way. Otherwise the (( )) (let command) syntax is used for arithmetic operations.

EXAMPLE

declare -i variable_name used for bash
typeset -i variable_name can be used to be compatible with ksh

(( n=5 + 5 ))
echo $n

Operators

The Bash shell uses the built-in test command operators to test numbers and strings, similar to C language operators.

EXAMPLE

<table>
<thead>
<tr>
<th>Equality:</th>
<th>Logical:</th>
</tr>
</thead>
<tbody>
<tr>
<td>== equal to</td>
<td>&amp; &amp; and</td>
</tr>
<tr>
<td>!= not equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>! not</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relational:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; greater than</td>
</tr>
<tr>
<td>&gt;= greater than, equal to</td>
</tr>
<tr>
<td>&lt; less than</td>
</tr>
<tr>
<td>&lt;= less than, equal to</td>
</tr>
</tbody>
</table>
2.6 The Bash Shell Constructs

Table 2.4 Bash Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Conditional statements</th>
<th>The if construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The then keyword is placed after the closing paren. An if must end with an endif. The new [[ ]] test command is now used to allow pattern matching in conditional expressions. The old [ ] test command is still available for backward compatibility with the Bourne shell. The case command is an alternative to if/else.</th>
</tr>
</thead>
</table>

**EXAMPLE**

The if construct is:

```bash
if  command
  block of statements
fi

if [[ expression ]] 
then
  block of statements
fi

if (( numeric expression ))
then
  block of statements
else
  block of statements
fi
```

The if/else/else if construct is:

```bash
if  command
  block of statements
else if command
  block of statements
else
  block of statements
fi

if [[ expression ]] 
then
  block of statements
elif [[ expression ]]
then
  block of statements
else
  block of statements
fi
```

The if/else construct is:

```bash
if  command
  block of statements
else
  block of statements
fi

if [[ expression ]] 
then
  block of statements
else if [[ expression ]] 
then
  block of statements
else
  block of statements
fi
```
Conditional statements

The case construct is:

```bash
case variable_name in
  pattern1
    statements
  pattern2
    statements
  pattern3
  esac

case "$color" in
  blue)
    echo $color is blue
  ;;
  green)
    echo $color is green
  ;;
  red|orange)
    echo $color is red or orange
  ;;
  *) echo "Not a match"
  ;;
esac
```

Loops

There are four types of loops: `while`, `until`, `for`, and `select`.

The `while` loop is followed by an expression enclosed in square brackets, a `do` keyword, a block of statements, and terminated with the `done` keyword. As long as the expression is true, the body of statements between `do` and `done` will be executed. The compound test operator `[[ ]]` is new with Bash, and the old-style test operator `[ ]` can still be used to evaluate conditional expressions for backward compatibility with the Bourne shell.

The `until` loop is just like the `while` loop, except the body of the loop will be executed as long as the expression is false.

The `for` loop is used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The `for` loop is followed by a variable name, the `in` keyword, a list of words, then a block of statements, and terminates with the `done` keyword.

The `select` loop is used to provide a prompt and a menu of numbered items from which the user inputs a selection. The input will be stored in the special built-in `REPLY` variable. The `select` loop is normally used with the `case` command.

The loop control commands are `break` and `continue`. The `break` command allows control to exit the loop before reaching the end of it, and the `continue` command allows control to return to the looping expression before reaching the end.
2.6 The Bash Shell Constructs

Table 2.4 Bash Shell Syntax and Constructs (continued)

<table>
<thead>
<tr>
<th>Loops (continued)</th>
<th><strong>EXAMPLE</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>while command</td>
<td>until command</td>
</tr>
<tr>
<td>do</td>
<td>do</td>
</tr>
<tr>
<td>block of statements</td>
<td>block of statements</td>
</tr>
<tr>
<td>done</td>
<td>done</td>
</tr>
<tr>
<td>while [[ string expression ]] do block of statements done</td>
<td></td>
</tr>
<tr>
<td>until [[ string expression ]] do block of statements done</td>
<td></td>
</tr>
<tr>
<td>while (( numeric expression )) do block of statements done</td>
<td></td>
</tr>
<tr>
<td>until (( numeric expression )) do block of statements done</td>
<td></td>
</tr>
<tr>
<td>for variable in word_list do block of statements done</td>
<td></td>
</tr>
<tr>
<td>select variable in word_list do block of statements done</td>
<td></td>
</tr>
<tr>
<td>for color in red green blue do echo $color done</td>
<td></td>
</tr>
<tr>
<td>select color in blue red green do show menu: 1) blue 2) red 3) green done</td>
<td></td>
</tr>
</tbody>
</table>

Functions

Functions allow you to define a section of shell code and give it a name. There are two formats, one from the Bourne shell, and the Bash version that uses the `function` keyword.

**EXAMPLE**

```bash
function_name() {
    block of code
}

function function_name {
    block of code
}------------------------

function lister {
    echo Your present working directory is `pwd`
    echo Your files are:
    ls
}
```
2.6.1 The Bash Shell Script

```bash
#!/bin/bash
# GNU bash versions 2.x
# The Party Program--Invitations to friends from the "guest" file
guestfile=~/.shell/guests
if [[ ! -e "$guestfile" ]]
then
    printf "${guestfile##*/} non-existent"
    exit 1
fi
export PLACE="Sarotini's"
(( Time=$(date +%H) + 1 ))
declare -a foods=(cheese crackers shrimp drinks "hot dogs" sandwiches)
for person in $(cat $guestfile)
do
    if [[ $person == root ]]
    then
        continue
    else
        # Start of here document
        mail -v -s "Party" $person "<" FINIS
        Hi $person! Please join me at $PLACE for a party!
        Meet me at $Time o'clock.
        I'll bring the ice cream. Would you please bring
        $foods[$n] and anything else you would like to eat?
        Let me know if you can make it.
        Hope to see you soon.
        Your pal,
        ellie@$(hostname)
    FINIS
    fi
    n=n+1
    if (( ${#foods[*]} == $n ))
    then
        declare -a foods=(cheese crackers shrimp drinks "hot dogs" sandwiches)
        n=0
    fi
done
printf "Bye..."
```
2.6 The Bash Shell Constructs

EXPLANATION

1. This line lets the kernel know that you are running a Bash shell script.
2. This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.
3. The variable `guestfile` is set to the full pathname of a file called `guests`.
4. This line reads: If the file `guests` does not exist, then print to the screen “guests non-existent” and exit from the script.
5. The built-in `printf` function displays only the filename (pattern matching) and the string “non-existent”.
6. An environment (global) variable is assigned and exported.
7. A numeric expression uses the output of the UNIX/Linux `date` command to get the current hour. The hour is assigned to the variable, `Time`.
8. A Bash array, `foods`, is defined with a list of elements.
9. An integer, `n`, is defined with an initial value of zero.
10. For each person on the guest list, except the user `root`, a mail message will be created inviting the person to a party at a given place and time, and assigning a food from the list to bring.
11. If the value in `$person` is `root`, control goes back to the top of the for loop and starts at the next person on the list.
12. The mail message is sent. The message body is contained in a here document.
13. The integer, `n`, is incremented by 1.
14. If the number of foods is equal to the value of the last number in the array index, the list is empty.
15. The array called `foods` is reassigned values. After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, ensuring that each person is assigned a food.
16. The variable `n`, which will serve as the array index, is reset back to zero.
17. This marks the end of the looping statements.