SECTION 6 Create Shell Scripts

In this section, you learn about the basic scripting elements and structures of the shell programing language.

Objectives

1. Use Basic Script Elements
2. Use Variable Substitution Operators
3. Use Control Structures
4. Use Advanced Scripting Techniques
5. Use Shell Functions
6. Learn About Useful Commands in Shell Scripts

Introduction

The Linux shell can control the system with commands and perform file operations or start applications. You can also create a file that includes several shell commands and start this file like a application.

This type of file is called a *shell script*. The following are several reasons why you need to understand and create shell scripts:

- You can automate many daily tasks with shell scripts. In many cases this increases speed and convenience in everyday work.
- The boot procedure and many other system functions are controlled by shell scripts. To understand and manipulate the system behavior, you need a basic understanding of shell programming.
- Shell programming is relatively easy to learn compared to other programming languages.
- A shell script runs on almost every UNIX-like operating system and does not need to be adapted to other platforms.

There are also some disadvantages to using shell scripts:

- Shell scripts are rather slow compared with other scripting languages.
- Shell scripts can use a lot of CPU power.

However, in most cases these disadvantages are not significant.

As you might have noticed, a Linux system offers different shell types. Shell scripts that are developed for one shell can sometimes be executed with a different shell, but this cannot be guaranteed.

For this reason, this section focuses on the Bash shell, which is the default shell in SLES 9.

As with all programing languages, shell scripting is learned best by actually writing code.

The exercises in this section include a description of a script that needs to be written. At the end of the section are the solutions to the exercises. We recommend attempting to create the script, and then comparing your script to the solution to understand the scripting concepts covered.

You can find all these scripts on the *3038 Course CD* in the directory **/exercises/section_6**. By using these scripts as a template, you can customize them to meet the needs of your production environment.

Although shell programing can be difficult at first, it becomes easier as you using the shell scripting language to automate tasks on your own system.

Objective 1 Use Basic Script Elements

The shell programming language is a powerful and complete programing language. Before you can start to create scripts, you need to become familiar with basic scripting techniques and elements.

In this objective, you learn the following about the basics of the shell programming language and simple shell scripts:

- Flow Charts for Scripts
- The Basic Rules of Shell Scripting
- How to Develop Scripts That Read User Input
- How to Perform Basic Script Operations with Variables
- How to Use Command Substitution
- How to Use Arithmetic Operations

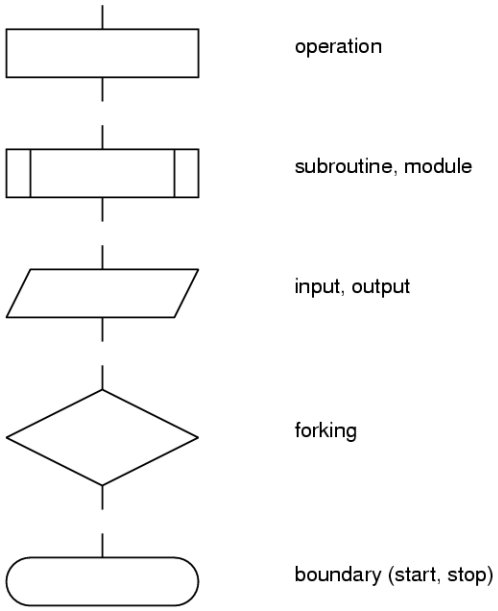
Flow Charts for Scripts

Programming elements of a script are often visualized by using program flow charts. Illustrating a program through a flow chart provides the following benefits:

- They force the author to lay down the steps the script should perform to achieve the desired goal, making it clearer which constructs need to be used.
- They provide a clear symbolic outline of the algorithm, which can be used as a guide during the programming process.

The following are typical symbols used to create flow charts:

Figure 6-1



The Basic Rules of Shell Scripting

Before writing your first shell script, you should consider a few points about scripting in general.

A shell script is basically an ASCII text file containing commands to be executed in sequence. To allow this, it is important that permissions for the script file are set to "r" (readable) and "x" (executable) for the user that runs it.

However, the execute permission is not granted by default to newly created file. To assign this permission, you need to use a command such as the following:

chmod +x script.sh

You can also run the script from another shell with a command such as the following:

sh script.sh

In this example, it is not necessary to make the script executable. On SLES 9, /bin/sh is a link to /bin/bash. It doesn't really matter whether you call the script with sh script.sh or bash script.sh.

Another important point is that the directory where the script is located must actually be in the user´s search path for executables.

A good way to deal with this is to create a /bin directory for scripts under each user´s home directory. Then you can add this directory to the user's search path by adding a line such as the following to your ~.bashrc:

```
export PATH=$PATH:~/bin
```

Otherwise, shell scripts must be started with the full pathname.

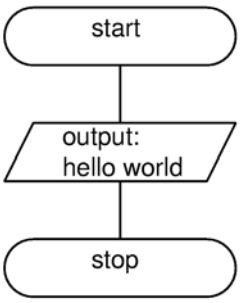
When naming script files, it is a good idea to add an .sh extension to the filename. This ensures that the file can easily be recognized as a shell script.

If you do not add the suffix, you need to make sure the filename is not identical to existing commands. For example, a common mistake is to name a script test.

The basic structure of a shell script can be illustrated with a simple program that does nothing more than print the message "Hello world."

The following is the flow chart for the script:

Figure 6-2

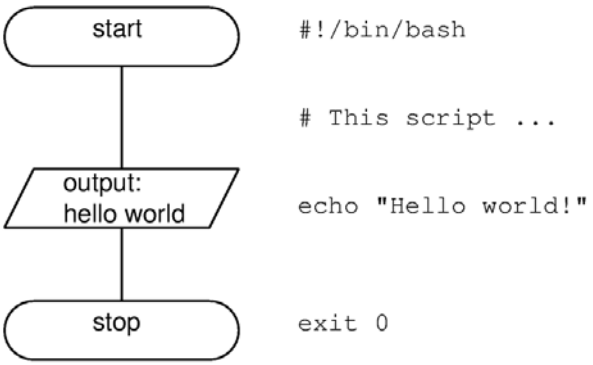


The script consists of three elements:

- The program start
- The action to print out “Hello world”
- The program stop

The following illustrates the 3 elements with the corresponding script code on the right:

Figure 6-3



Before looking closer at each of the 3 elements, you need to understand that the general rules for creating shell scripts, as explained in this section, can be applied to any conceivable script.

The following describes the 3 elements of the script:

- **Start.** The first line of any shell script must be the *shebang* (such as **#!/bin/bash**). This line specifies the shell program to be called to execute the script. As with any other program, a subshell is started to run the script.

  The script’s start section should also include a comment describing what the script does. A comment is introduced with a **#** character in shell scripts.

  It is also a good idea to include the name of the author, the date, and the version number of the script. Also, any variables and functions used in the script should be defined at the top of the script.

- **Commands.** The sample script above includes the echo command as the only one executed (to print the “Hello world” greeting). Shell scripts in general rely on the echo command as the most common solution to display information on the screen.

- **Stop.** Before the script ends, it might be necessary to do some cleanup. For example, you might want to remove any temporary files created by the script.

  As the very last step, you should define the script´s exit status with an exit value. This informs the parent process how the script was terminated. The exit status as returned by the script can be queried afterward with echo $?.

Every script that you write should use this basic structure.

Exercise 6-1 Produce Output from a Script

Do the following:

1. Write a script that outputs “Hello world.” Use the following command in the script:

   echo -e “\aHello\nworld”

2. Find out the purpose of the **\a**, the **\n** and the **-e** options (try accessing the man pages).

3. Compare your solution with the script at the end of the section.

This script is also available as **hello.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

How to Develop Scripts That Read User Input

One way to create scripts that read user input is to use the command **read**. The read command takes a variable as an argument and stores the read input in the variable. The variable can then be used to process the user input.

The following example reads user input into the variable with the name VARIABLE:

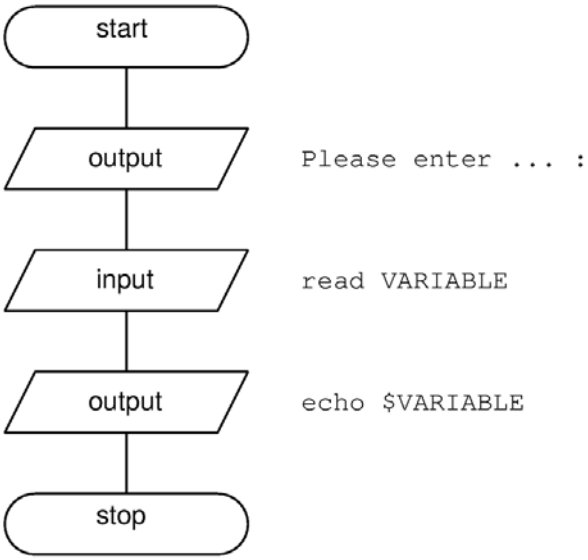
```
read VARIABLE
```

The script pauses at this point, waiting for user input until the Enter key is pressed. To tell the user to enter something, you need to print (echo) a line with some information, such as the following:

```
echo "Please enter a value for the variable:"
read VARIABLE
```

The following flow chart illustrates the structure of a script that reads user input:

Figure 6-4



First, the script produces some output with echo to ask the user to enter something. Then the read command waits until the input is provided to store it in the variable VARIABLE. At the end the content, the variable is printed out with echo.

Exercise 6-2 Read User Input

Do the following:

1. Create a simple shell script that prompts the user to enter her first and last name, and then greets the user with her full name.
2. Compare your solution with the script at the end of the section.

This script is also available as **name1.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

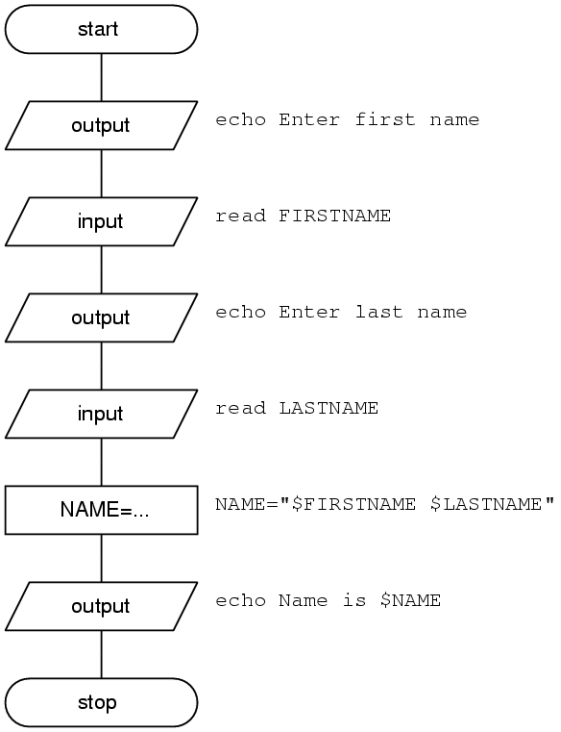
(End of Exercise)

How to Perform Basic Script Operations with Variables

In this part of the section, you learn how to uses variables in shell scripts.

The following flowchart and script show how a string value can be assigned to a variable:

Figure 6-5



You want to read the user's first and last name and then print both names to the screen. However, this time you create a variable called NAME, which holds both the first and the last name.

The following is an interesting line in the script:

```
NAME=$FIRSTNAME $LASTNAME
```

This line shows how you can combine two variables, in this case, FIRSTNAME and LASTNAME, and assign the combined value to another variable, in this case, NAME.

In this example, you can also see another rule of the variable handling in shell scripts. If you assign a value to a variable, you use just the name of the variable, in this case, NAME=.

If you want to use the value of a variable, put a $ before the name, in this case, $FIRSTNAME.

It is often useful to assign a default value to a variable. This might prevent errors, if the user has entered a value that cannot be interpreted in a meaningful way.

If the variable FIRSTNAME is empty, the default value FLORIAN is used instead, as in the following:

```
NAME=${FIRSTNAME:=FLORIAN}
```

Exercise 6-3 Simple Operations with Variables

Do the following:

1. Modify your script from Exercise 6-2 so that it reads the user's first and last name, combines both in one variable, and outputs the variable.
2. Compare your solution with the script at the end of the section.

This script is also available as **name2.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

How to Use Command Substitution

The term *command substitution* basically means that the output of a command is used in a shell command line or a shell script.

In the following example, the output of the command date is used to generate the output of the current date:

```
#!/bin/bash

echo "Today is `date +%m/%d/%Y`"
```

An important thing to remember is that the command date +%m/%d/%Y is included in backticks (` ... `).

Instead of printing the output of a command to the screen with echo, it can also be assigned to a variable, as in the following:

```
#!/bin/bash

TODAY=`date +%m/%d/%Y`
echo "Today is $TODAY"
```

In this case, the output of date is assigned to the variable TODAY, and then TODAY is printed to the screen with echo. Make sure that there are no spaces before or after the equal sign.

Exercise 6-4 Use Command Substitution

Do the following:

1. Create a shell script that outputs the current login name and the current working directory.

   The output of the commands **whoami** and **pwd** should be read into variables with the variables printed to the screen.

2. Compare your solution with the script at the end of the section.

This script is also available as **info.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

How to Use Arithmetic Operations

Shell scripts often use values assigned to variables for calculation. There are several ways to implement this.

The Bourne shell is limited in this regard, but it can perform such operations by relying on external commands (such as expr).

The Bash shell comes with built-in support for arithmetic operations, but there are some limitations to this as well. Specifically, the arithmetic capabilities of Bash are limited in the following ways:

- Only operations with whole numbers (integers) can be performed.
- All values are signed 64-bit values. Thus, possible values range from -2^{63} to $+2^{63}-1$.

So even when using Bash, you might need to use external commands, such as bc for floating-point calculations.

The following paragraphs list all the possible methods and formats for arithmetic operations. All of them use this sample operation:

```
A=B+10
```

- **Use the external command expr (Bourne shell compatible)**

  ```
  A=`expr $B + 10`
  ```

  Since an external command is used, this method will also work with the Bourne shell. Scripts using external commands will always perform slower than those relying on built-in commands.

- **Use the Bash built-in command let**

  ```
  let A="$B + 10"
  ```

  In Bash, you can use the let command to perform an arithmetic expression.

- **Use arithmetic expressions inside parentheses or brackets (two different formats)**

  ```
  A=$((B + 10))
  ```

  or

  ```
  A=$[B + 10]
  ```

  Arithmetic expressions can be enclosed in double parentheses or in brackets for expansion by Bash. Both $((. . .)) and $[. . .] are possible, but the latter is considered deprecated and should be avoided.

- **Use the built-in command declare**

```
declare -i A
declare -i B
A=B+10
```

This declares a variable as an integer.

If all the variables involved in a calculation have previously been declared as integers through declare -i, arithmetic evaluation of these variables happens automatically when a value is assigned to them.

This means that the variable B, for example, does not have to be prefixed with the $ to be evaluated.

With the **expr** command, only the following five operators are available: + , - , * , / , and %. Additional operators (which are identical to those of the C programming language) can be used with all of the above Bash formats.

For a complete list, consult the man page for bash.

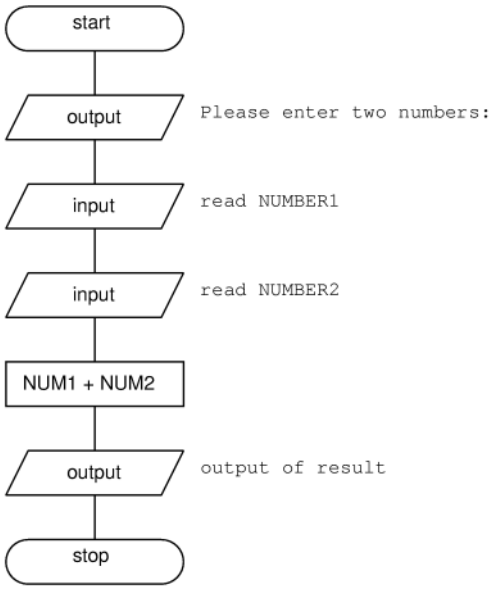
It makes sense to limit yourself to using one of the described possibilities. As far as Bash is concerned, a good choice might be to only use the declare command, since it makes the best use of the available features.

Exercise 6-5 Use Arithmetic Operations

Do the following:

1. Review the following flowchart:

Figure 6-6



2. Write a shell script that reflects the above flowchart.
3. Modify the script to use the other fundamental arithmetic operations (subtraction, multiplication, division).
4. Find out what happens if
   - The user enters a word for each number.
   - The user enters nothing (presses **Enter**) at each prompt.
5. Compare your solution with the script at the end of the section.

This script is also available as **sum.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Objective 2 Use Variable Substitution Operators

In Bash, you can use special variable substitution operators to assign different values to variables without having to rely on external commands.

For example, these special substitution operators allow changing variables by deleting certain patterns in their values and returning the rest.

They also allow you to set a default for a variable for situations where no value can be assigned to it.

The following variable substitutions are possible:

Table 6-1

Substitution Operator	Description
${variable-value}	Returns value if the variable does not exist.
${variable=value}	Assigns value to the variable and returns value if the variable does not exist.
${variable+value}	Returns value if the variable exists.
${#variable}	Returns the number of characters in the value of variable.
${variable#pattern}	Deletes the shortest part matched by pattern from the beginning of the variable's value and returns the rest.
${variable##pattern}	Deletes the longest part matched by pattern from the beginning of the variable's value and returns the rest.
${variable%pattern}	Deletes the shortest part matched by pattern from the end of the variable's value and returns the rest.
${variable%%pattern}	Deletes the longest part matched by pattern from the end of the variable's value and returns the rest.

The substitution operators returning or setting a default value (- , = , and +) can also be prefixed with a colon so that substitution happens if the variable does not exist of if it exists but has a null value (is empty).

The following are some examples of how to use the substitution operators:

```
tux@DA1:~> echo $VAR

tux@DA1:~> echo ${VAR-value}
value

tux@DA1:~> echo $VAR

tux@DA1:~> echo ${VAR=value}
value

tux@DA1:~> echo $VAR
value

tux@DA1:~> VAR=
tux@DA1:~> echo ${VAR=value}

tux@DA1:~> echo ${VAR:=value}
value

tux@DA1:~> echo $VAR
value

tux@DA1:~> echo ${VAR+VaLue}
VaLue
```

Exercise 6-6 Use Variable Substitution

Do the following:

1. Write a script that asks the user for a filename, and then performs a search for that filename using the command **find**.

   Use a variable substitution to assign a default value for the filename (such as *.bak) in case the user enters nothing.

2. Compare your solution with the script at the end of the section.

This script is also available as **find.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Objective 3 Use Control Structures

Using the scripting techniques you have learned so far, you can only develop scripts that run sequentially from the beginning to the end.

In this objective, you learn how to use control structures to make the execution of parts of your script dependent on certain conditions or to repeat script parts.

To use control structures, you need to know how to do the following:

- Create Basic Branches With the if Command
- Build Multiple Branches With a case Statement
- Create Loops Using the while and until Commands
- Process Lists with the for Loop
- Interrupt Loop Processing

Create Basic Branches With the if Command

You can use the if command to perform certain actions in your script that depend on a condition.

The following is the basic usage of the if command:

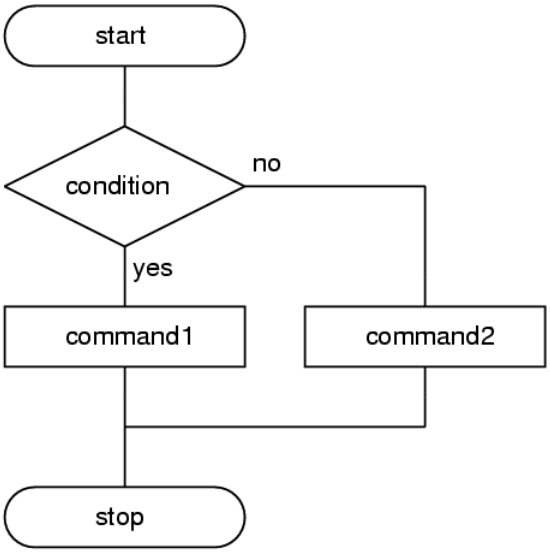
```
if condition
then
  commands
fi
```

The if statement can be extended with an optional else statement, as in the following:

```
if condition
then
  command1
else
  command2
fi
```

In a program flow chart, a branch created with an if statement can be represented like the following:

Figure 6-7



A branch of this type must begin with if and end with fi. Command1 is only executed if the condition is true.

If the return code of a command is used as condition, the exit code zero (success) represents true. If the exit status is not zero or the condition is not true, the shell goes to the end of the branch or, if an else statement is present, to the else statement.

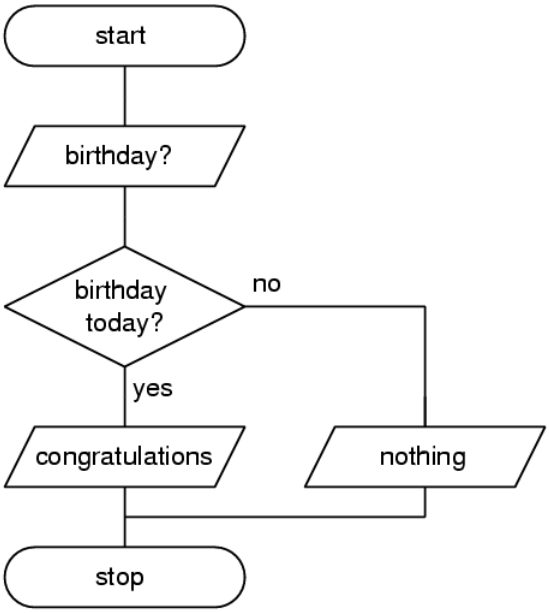
When you use these control structures in a shell script, individual commands (such as if, then, and fi) must follow immediately after a command separator.

In the above case, the separator is a new line. The separator could also be a semicolon, which would allow you to enter the same if statement as one command, as in the following:

```
if condition; then commands; fi
```

The following example uses a sample script to explain how an if branch works:

Figure 6-8



This script asks the user to enter his date of birth; if that happens to be today, the script congratulates him on his birthday. It does nothing if his birthday is another day.

There are a number of items to consider when writing this script. From the flow chart, it should be obvious that the script consists of 2 basic steps:

- Prompt the user to enter the date of birth.
- Compare the date as entered by the user with the current date. If the dates are the same, the user sees “congratulations.” If they are not equal, nothing appears.

The branch is the actual mechanism that compares the current date and the date of birth.

Before the comparison can be performed, both dates must be available in the same format. The user should be asked to specify the date of birth in a suitable format.

You need to know the format in which the system obtains the current date. The obvious choice to get a date string is with the command date.

The command date + %m-%d returns the current date in the form month-day, as in the following:

```
date + %m-%d
06-21
```

This format should also be used for the birth date the user is requested to enter:

```
echo "Please enter your date of birth (YYYY-MM-DD, for instance
1978-06-21): "
read BIRTHDAY
```

The second part of the listing consists of several items. To check if the user´s birthday is today, 2 dates must be compared: the birthday and the current date.

The user´s birthday is stored in the variable BIRTHDAY. The current date must also be stored in a variable for the comparison. This can be done using command substitution, as in the following:

```
TODAY=`date + %m-%d`
```

A closer examination of the comparison reveals that the values in the variables cannot be compared with each other (BIRTHDAY: 1973-12-21, TODAY: 09-24). Therefore, the dates must be compared without the year.

To do this, the variable substitutions of the Bash shell can be used to truncate the year from the date. The first part of the script should look like the following:

```
#!/bin/bash
echo "Please enter your date of birth (YYYY-MM-DD, for instance
1978-06-21): "
read BIRTHDAY
BIRTHDAY=${BIRTHDAY#*-}
TODAY= date + %m-%d
```

Now you can compare the two values with the help of an if branch. Most variables are compared using the test command. The test command is followed by a string condition such as
test $VARIABLE1 = $VARIABLE2.

If the condition is met (if the value of VARIABLE1 is identical to the value of VARIABLE2), test returns a zero to indicate success.

So the second part of the shell script could look like the following:

```
if test "$BIRTHDAY" = "$TODAY"
then
  echo "Tada! Happy birthday to you! Nice presents awaiting you ..."
else
  echo "Sorry to disappoint you, no presents today ..."
fi
```

Finally, you want the script to use the exit command to finish with a certain exit status, which depends on whether today is the user´s birthday. This is implemented by defining yet another variable, as in the following:

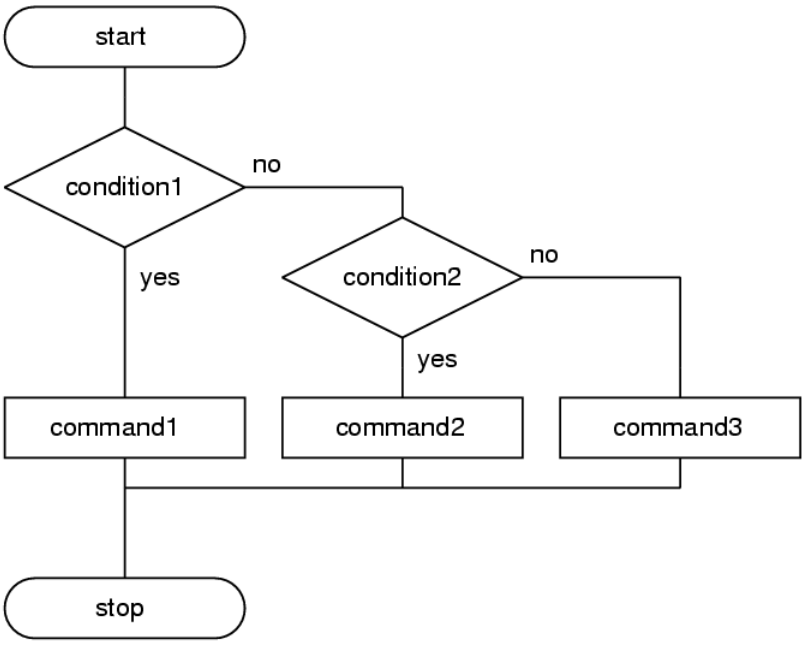
```
if test "$BIRTHDAY" = "$TODAY"
then
  echo "Tada! Happy birthday to you! Nice presents awaiting you ..."
  STATUS=0
else
  echo "Sorry to disappoint you, no presents today ..."
  STATUS=1
fi
exit $STATUS
```

Several if branches are often nested in each other. The command elif, which represents a more compact way of writing the command sequence else if, is useful for the following kind of structures:

```
if condition1
then
  command1
elif condition2
  command2
else
  command3
fi
```

The elif command is illustrated in the following:

Figure 6-9



There are several ways to use the Bash shell to successively execute several commands. This includes using the separators && and ||, which make it possible to execute a second command depending on the success or failure of the first, as in the following:

```
command1 && command2
command1 || command2
```

- The && separator executes command2 if the command1 exits with success.
- The || separator executes command2 if the command1 exits with a failure.

These separators can also be understood as short forms of an if branch.

This means that the following structure:

```
if test -e file
then
  . file
fi
```

can be condensed into the following command line:

```
test -e file && . file
```

Whenever the comparison is a simple one as in the example above, you can replace the relatively complex if. . .then. . .fi structure with a command line that uses && or || to chain the commands.

Exercise 6-7 Use the if Command

Do the following:

1. Write a shell script that checks for the existence of a given file, and if the file is executable.

   A message should be displayed for each of the following scenarios:

   - The file does not exist.
   - The file exists.
   - The file exists and is executable.

   You can use the command **test -x** to check whether a file is executable.

2. Compare your solution with the script at the end of the section.

This script is also available as **file_check.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Build Multiple Branches With a case Statement

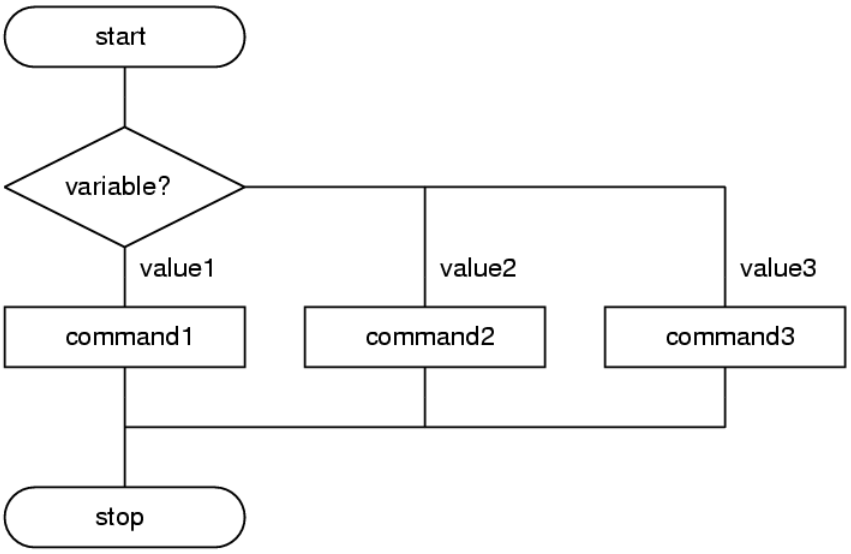
You can create multiple branches with case. In a case statement, the expression contained in a variable is compared with a number of expressions, and a command is executed for each expression matched.

A case statement has the following structure:

```
case $variable in
  expression1) command1;;
  expression2) command2;;
esac
```

In a flow chart, multiple branching with case looks similar to a simple branch created with if:

Figure 6-10



The following is an example of how a multiple branch works:

```
#!/bin/bash
cat << EOF
  Name me an animal and I will tell you how many legs it has!
EOF

read CREATURE
case "$CREATURE" in
  dog | cat | mouse ) echo "A $CREATURE has 4 legs."
  ;;
  bird | human | monkey ) echo "A $CREATURE has 2 legs."
  ;;
  spider ) echo "A $CREATURE has 8 legs."
  ;;
  fly ) echo "A $CREATURE has 6 legs."
  ;;
  * ) echo "I haven t the faintest idea how many
      legs a(n) $CREATURE has."
  ;;

esac
exit 0
```

This script prompts the user to enter the name of an animal. The name is then stored in a variable and compared with a number of possible matches. For the matches found, the script tells the user how many legs the animal has.

To allow for several expressions to be matched within one and the same branch, several expressions can be listed on one line with a | symbol as a separator.

The user input is read and assigned to the CREATURE variable.

The case statement then compares this value against each of the expressions provided as alternatives. For instance, if the user enters cat, the script prints the matching sentence that says that this animal has four legs.

The asterisk (*) is often used as the last expression to be evaluated to cover all cases not matched by the other alternatives. The corresponding message states that the number of legs is not known.

It is important that the expressions provide for an exact match of any allowable expression. For instance, if someone entered Dog instead of dog, the script will not know the number of legs for this strange kind of animal.

For this reason, it is useful to supply the possible alternatives beforehand, as in the following:

```
...

case "$CREATURE" in
  [dD]og | [cC]at | [mM]ouse )

...
```

You can provide such alternatives in brackets, in the same way as the shell´s filename expansion mechanism.

Exercise 6-8 Use the case Command

Do the following:

1. Create an example (not a complete script) to show how a script can use a case statement to process a user's answer to a Yes/No question. Include the responses as “yeah” and “nope.”
2. Compare your solution with the example at the end of the section.

This example t is also available as **yes_no.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Create Loops Using the while and until Commands

The purpose of a loop is to test a certain condition and to execute a given command while the condition is true (while loop) or until the condition becomes true (until loop).

The following is the structure of a while loop:

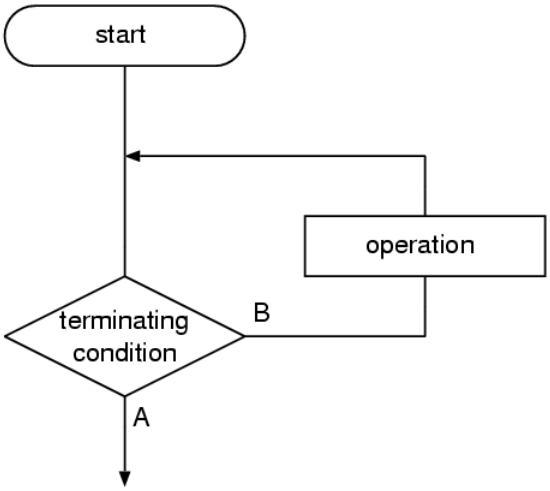
```
while condition
do
  commands
done
```

The following is a structure of until loop:

```
until condition
do
 commands
done
```

The following illustrates the loop constructs:

Figure 6-11



These loops actually rely on the exit status of a terminating condition: a while loop remains operative as long as the condition's exit status is zero (path B in the flow chart), but an until loop is terminated if the status is zero (path A in the flow chart).

A while loop is terminated when the exit status becomes nonzero (when the condition is not true), but an until loop is operative as long as the status is nonzero.

Exercise 6-9 Use the while and until Commands

Do the following:

1. Create a script that performs a simple while loop 100 times. In every iteration, the number of the current iteration should be printed to screen.
2. Write a second script which uses until instead of while.
3. Compare your solution with the scripts at the end of the section.

These scripts are also available as **counter1.sh** and *counter2.sh* in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Process Lists with the for Loop

The purpose of a for loop is to process a list of elements. It has the following syntax:

```
for variable in element1 element2 element3
do
  commands
done
```

A for loop executes the given commands once for every element on the list, and the value of the variable matches one list element with each loop iteration. The list itself is often created through command substitution.

If the for command is not accompanied by a list of elements, the loop will be executed with the contents of the variables $1, $2, $3, and so on. These elements represent the command line parameters that are passed to the script.

As with similar constructs, a command separator must immediately precede the do and done parts of the for loop. This means that a for loop can be entered in 2 different ways:

```
for i in 1 2 3 4 5 6 7 8
do
  ping -c1 DA$i
done
```

In the example above, the command separator is a line break and the loop is started only after entering the final done.

You could use a semicolon as a separator instead, In this case, the same loop looks like the following:

```
for i in 1 2 3 4 5 6 7 8; do ping -c1 DA$i; done
```

If you want to use a range of numbers in your for loop, you can use the following C-Style syntax:

```
LIMIT=10

for ((a=1; a <= LIMIT ; a++))
do
  echo -n "$a "
done
```

In this example, the variable a is a count from 1 to 10. The for expression contains the following 3 elements:

- **a=1.**This determines the start value for a.
- **a <= LIMIT.** This is the condition when the for loop should be terminated; in this case, when the a variable reaches the value determined by the LIMIT variable (in this example, the value 10).
- **a++.** This command adds 1 to the a variable for every pass of the for loop.

Exercise 6-10 Use the for Loop

Do the following:

1. Create a shell script that renames all files in the current directory with uppercase letters transformed to lowercase.

   Hints:

   - Use the command **find . -type f -maxdepth 1** to find all files in the current directory.
   - You can use the command **tr [A-Z] [a-z]** to convert uppercase letters to lowercase.
   - If you don't know how to start, have a brief look at the solution at the end of the section.
   - Test your script in a directory that does not contain important files.

2. Compare your solution with the script at the end of the section.

This script is also available as **lowercase1.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Interrupt Loop Processing

Use the continue command to exit from the current iteration of a loop (while, until, for, and select) and resume with the next iteration of the loop.

This allows a script to test for an additional condition with each iteration without stopping completely (as a result of the terminating condition becoming true, for instance).

The following is an example of using the continue command:

```
for FILE in  ls *.mp3
do
  if test -e /MP3/$FILE
  then
    echo "The file $FILE exists."
    continue
  fi
  cp $FILE /MP3
done
```

This script writes a backup copy of all files ending with .mp3 to the directory /MP3/ unless there is already a file with the same name in that directory. If there is, the script prints a message stating that the file already exists and exits from the current loop iteration.

The break command is another way to introduce a new condition within a loop. Unlike continue, it causes the loop (not the current loop iteration) to be terminated completely if the condition is met.

Exercise 6-11 Interrupt Loop Processing

Do the following:

1. Modify the script from Exercise 6-10 so that existing files in the current directory are not overwritten.

   Use continue to interrupt the iteration over the files in the directory if a file with the target name already exists.

2. Compare your solution with the script at the end of the section.

This script is also available as **lowercase2.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Objective 4 Use Advanced Scripting Techniques

In this objective, you learn the following advanced scripting techniques that can help you solve common problems of script development:

- Use Shell Functions
- Read Options with getopts

Use Shell Functions

Sometime you need to perform a task multiple times in a shell script. Instead of writing the same code again and again, you can use functions.

Shell functions act like script modules because they make an entire script section available with a single name. Shell functions are normally defined at the beginning of a script. You can store several functions in a file and include this file whenever the functions are needed.

The following is the basic syntax of a function:

```
functionname () {
  commands
  commands
}
```

The following generates a function with the function command:

```
function functionname {
  commands
  commands
}
```

The function name can be composed of any regular character string that then can be used to call the function.

The following is a simple function that creates a directory and then changes to that directory:

```
# mcd: mkdir + cd; creates a new directory and
# changes into that new directory right away

mcd (){
  mkdir $1
  cd $1
}
```

After having been created, this function can be called in a shell scripts, as in the following:

```
...
mcd directory
...
```

The parameter directory is called an *argument*. Within a function, arguments can be accessed with the variables $1, $2, $3, and so on, depending on the number of arguments passed to the function.

The following function can be used to create a pause in a script. The script resumes only after the Enter key is pressed:

```
# pause: causes a script to take a break

pause (){
  echo "To continue, hit RETURN."
  read q
}
```

You can also create functions that stop their processing from within, similar to exiting a loop (iteration) with the commands break and continue.

To exit a function, use the command return. If return is called without an argument, the return value of the function is identical to the exit status of the last command executed in that function.

Otherwise, the return value is identical to the one supplied as an argument to return.

Exercise 6-12 Use Shell Functions

Do the following:

1. Review the following shell function:

```
# Prompt the user to answer with "yes" or "no.
# The question itself is supplied as an argument
# when calling the function, for example:
# "yesno Do you want to continue?"

yesno (){
  while true
  do
    echo "$*"
    echo "Please answer by entering (y)es or (n)o:"
    read ANSWER
    case "$ANSWER" in
      [yY] | [yY][eE][sS] )
         return 0
         ;;
      [nN] | [nN][oO] )
         return 1
         ;;
      * )
         echo "I cannot understand you over here."
         ;;
     esac
  done }
```

   This function asks the user to enter y or n. Depending on the answer, the function returns 0 or 1. If the answer is wrong, an error message is displayed.

   The command echo “$*” is used to print a question, which is passed as a parameter to the function.

2. Use the above yesno function to write a script that lets the system administrator delete user accounts.

   The script should prompt for the account to delete, and then asks whether the user's home directory should also be deleted.

   If the question is answered with **no**, the script should change the user and group ownership of the corresponding home directory to root.

   After doing so, the script should use the yesno function again to ask whether the administrator really wants to delete the account.

   Use the commands userdel and chown in the script to perform the necessary tasks.

   You can assume that the home directory of the user is always located in /home and that the name of the directory is the same as the login name of the user.

3. Test your solution by adding a user account (enter **useradd -m tux2**) and deleting it.

4. Compare your solution with the script at the end of the section.

This script is also available as **userdel1.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Read Options with getopts

With the shell built-in command getopts, you can extract the options supplied to a script on the command line. The shell interprets command-line arguments as command options only if they are prefixed with a - (the default when using the shell interactively).

This makes it possible to place options in different positions on the command line and to supply them in an arbitrary order.

This means that the following command:

cp -dpR *.txt texts/

achieves the same thing as the command

cp -R *.txt -d texts/ -p

getopts recognizes options in the same way. The following is the getopts syntax:

getopts *optionstring variable*

The ***optionstring*** describes all options to be recognized. For instance, getopts abc declares a, b, and c as the options to be processed.

If a parameter is expected for the option (such as -m maxvalue), the corresponding option must be followed by a : in the string (as in getopts m:).

The option string is followed by a ***variable*** to which all the command-line options specified are assigned as a list.

The getopts command is mostly frequently used in a while loop together with case to define which command to execute for a given option, as in the following:

```
while getopts abc: variable
do
  case $variable in
    a ) echo "The option -a was used." ;;
    b ) echo "The option -b was used." ;;
    c ) option_c="$OPTARG"
        echo "Option c has been set." ;;
  esac
done

echo $option_c
```

If the option -a or -b is used, the script prints out a message that the corresponding option was used. If the option -c *value* is used, the value is assigned to the variable option_c, which is printed to the screen at the end of the script.

The parameter of an option can be accessed with the variable OPTARG.

Exercise 6-13 Use the getopts Command

Do the following:

1. Modify the script from Exercise 6-12 so that it does not prompt the user for input. Instead, the script should use the following options:
   - **-u *username*.** This option determines the user which shall be deleted.
   - **-r.** If this option is set, the home directory should be removed. If this option is not set, the owner of the home directory should be set to root.
2. Test you solution by adding a user account (enter **useradd -m tux2**) and deleting it.
3. Compare your solution with the script at the end of the section.

This script is also available as **userdel2.sh** in the directory **/exercises/section_6** on your *3038 Course CD*.

(End of Exercise)

Objective 5 Learn About Useful Commands in Shell Scripts

You can use external commands in shell scripts to perform certain tasks. In this objective, you learn how to

- Use the cat Command
- Use the cut Command
- Use the date Command
- Use the echo Command
- Use the grep and egrep Commands
- Use the sed Command
- Use the test Command
- Use the tr Command

Use the cat Command

When combined with the here operator (**<<**), the cat command is a good choice to output several lines of text from a script. In interactive use, the command is mostly run with a filename as an argument, in which case cat prints the file contents on standard output.

Use the cut Command

You can use the cut command to cut out sections of lines from a file, so only the specified section is printed on standard output.

The command is applied to each line of text as available in a file or on standard input. You can use **cut -f** to cut out text fields. **cut -c** works with the specified characters.

You can specify single sections (characters or fields) or several sections. The default delimiter to separate fields from each other is a tab, but you can specify a different field separator with the -d option.

The following are some examples of using cut:

```
cut -d : -f1 /etc/passwd
root
bin
daemon
lp
mail
news
```

The above command specifies that the field separator should be a colon. In every line of /etc/passwd, the field that comes before the first colon is taken and printed to stdout:

```
ls -l somedir/ | cut -c 35- | sort -n
687 Sep 20 17:06 file2
2199 Sep 20 17:05 file1
6593 Sep 20 17:06 file3
```

The above command takes the output of the ls command and cuts out everything from the thirty-fifth character. This is piped to sort, so the final output is sorted according to file size.

Use the date Command

You can use the date command whenever there is a need to obtain a date or time string for further processing by a script. Without any options specified, the command´s output looks like the following:

```
date
Fre Sep 03 14:18:12 CEST 2004
```

The date command lets you change the output format in almost every detail. With the -I option (as in the following), date prints the date and time in ISO format (which is the same as if the options had been +%Y-%m-%d):

```
date -I
2004-09-03

date +%m-%d %H:%M
09-03 14:19

date +%D, %r
09/03/02, 02:19:58 PM

date +%d.%m.%y
03.09.02

date +%d.%m.%Y
03.09.2004

date +%e.%-m.%y,   %l.%M   %p
3.9.02, 2.20 PM

date +%A, %e. %B   %Y
Friday, 3. September 2004
```

To view a list with all the possible format options for date, see **man date**. In any case, you should be able to customize the output to exactly match the requirements of your script.

Use the echo Command

The echo command, which exists both as a shell built-in command and as an external command, prints text lines on standard output. A line break is inserted automatically after each line. When called with the -e option, echo accepts a number of additional options.

The following are some of the special sequences recognized by echo when run with the -e option:

- **\a.** Outputs an alert (sounding the bell). This does not work in the KDE Konsole.
- **\c.** Do not add a new line at the end of the output.
- **\n.** Add a new line (line break).

The cat command is preferred over echo to output a text file or several lines of text.

Use the grep and egrep Commands

The command grep and its variant egrep are used to search files for certain patterns, and use the following syntax:

grep *searchpattern filename ...*

The command prints lines that contain the given search pattern. You can specify several files, in which case the output will print the matching line and the corresponding filenames.

Several options are available to specify that only the line number should be printed, for instance, or that the matching line should be printed together with leading and trailing context lines.

Search patterns can be supplied in the form of regular expressions, although the bare grep command is limited in this regard.

To search for more complex patterns, use the egrep command, which accepts extended regular expressions. As a simple way to deal with the difference between the two variants, make sure you use egrep in all of your shell scripts.

The regular expressions used with egrep need to be in accordance with the standard regex syntax.

To avoid having special characters in search patterns interpreted by the shell, enclose the pattern in quotation marks, as in the following:

```
tux@DA1:~> egrep (b|B)lurb file*
bash: syntax error near unexpected token  |

tux@DA1:~> egrep "(b|B)lurb" file*
file1:blurb
file2:Blurb
```

Use the sed Command

The sed program is a stream editor, an editor used from the command line rather than interactively. sed performs text transformations on a line-by-line basis.

You can specify sed commands either directly on the command line or in a special command script loaded by the program on execution.

The following is the syntax for the sed command:

sed *editing-command filename*

The available editing commands are single-character arguments such as the following:

- **d:** Delete
- **s:** Substitute (replace)
- **p:** Output line
- **a:** Append after

As with other commands, the output of sed normally goes to standard output, but it can also be redirected to a file.

Each sed command must be preceded by an exact address or address range specifying the lines to which the editing command applies.

Apart from the single-character commands for text transformations, you can also specify options to influence the overall behavior of the sed program.

The following are some important command-line options for sed:

- **-n, --quiet, --silent.** By default, sed will print all lines on standard output after they have been processed. This option suppresses the output so sed only prints those lines for which the **p** editing command has been given to explicitly re-enable printing.
- **-e *command1* -e *command2* ....** This option is necessary when specifying two or more editing commands. It must be inserted before each additional editing command.
- **-f *filename*.** With this option, you can specify a script file from which sed should read its editing commands.

For many editing commands, it is important to specify the exact line or lines that should be processed by the command. One of the more frequently used address labels is $, which stands for the last line.

The following are 2 examples of the sed command:

- **sed -n ‘1,9p’ somefile**

  This command prints only lines 1 through 9 on stdout.

- **sed ‘10,$d’ somefile**

  This command deletes everything from line 10 to the end of the file and also prints the first 9 lines of somefile.

You can use a regular expression to define the address or address range for an editing command. Regular expressions must be enclosed in forward slashes. If an address is defined with such an expression, sed processes every line that includes the given pattern.

The following is an example of using regular expressions:

sed -n ‘/Murphy.*/p’ somefile

This example prints all lines that have the pattern Murphy.* in them.

If you want sed to perform several editing commands for the same address, you need to enclose the commands in braces, as in the following:

sed ‘1,10{*command1* ; *command2*}’

The following lists the most important editing commands available for sed:

Table 6-2

Command	Exampleo	Editing Action
d	sed 10,$d *file*	Delete line.
a	sed ‘a*text**text*’ *file*	Insert text before the specified line.
i	sed ‘i*text**text*’ *file*	Replace specified lines with the text.
c	sed ‘2000,$c*text* ‘ *file*	Replace specified lines with the text.
s	sed s/*x*/*y*/*option*	Search and replace. The search pattern *x* is replaced with pattern *y*. The search and the replacement pattern are regular expressions in most cases and the search and replace behavior can be influenced through various options.
y	sed y/abc/xyz/	(yank) Replace every character from the set of source characters with the character that has the same position in the set of destination characters.

You can use the following options with the s command (search and replace):

- **I.** Do not distinguish between uppercase and lowercase letters.
- **g.** Replace globally wherever the search pattern is found in the line (instead of replacing only the first instance).
- **n.** Replace the nth matching pattern only.
- **p.** Print the line after replacing.
- **w.** Write the resulting text to the specified file rather than printing it on stdout.

The following are some examples of using the **s** command:

- **sed 's/:/ /' /etc/passwd**

  This command replaces the first colon in each line with a space.

- **sed 's/:/ /g' /etc/passwd**

  This command replaces all colons in all lines with a space.

- **sed 's/:/ /2' /etc/passwd**

  This command replaces only the second colon in each line with a space.

- **sed -n 's/\([aeiou]\)/\1\1/Igp'**

  This command replaces all single vowels with double vowels. The example shows how matched patterns can be referenced with "\1" if the search pattern is given in parentheses (which have to be escaped). The I option ensures that sed ignores the case.

  The g option causes characters to be replaced globally. The p option tells sed to print all lines processed in this way.

Use the test Command

The test command exists both, as a built-in command and as an external command. It is used to compare values and to check for files and their properties (whether a file exists, whether it is executable, and so on).

If a tested condition is true, test returns an exit status of 0; if the condition is not true, the exit status is 1. In shell scripts, ttest is used mainly to declare conditions to influence the operation of loops, branches, and other statements.

The following is the test syntax:

test *condition*

You can use the test command to do the following:

- Testing whether a file exists. Some of the available options are:

Table 6-3

Option	Description
-e	File exists
-f	File exists and is a regular file
-d	File exists and is a directory
-x	File exists and is an executable file

- Comparing 2 files. Some of the available operators are:

Table 6-4

Option	Description
-nt	Newer than
-ot	Older than
-ef	Refers to the same inode (such as in the case of a hard link)

- Comparing 2 integers. The available operators are:

Table 6-5

Option	Description
-eq	Equal
-ne	Not equal
-gt	Greater than
-lt	Less than
-ge	Greater than or equal
-le	Less than or equal

- Testing strings. The available operators are:

Table 6-6

Option	Description
test -z *string*	Exit status is 0 (true) if the string has zero length (is empty).
test *string*	Exit status is 0 (true) if the string has nonzero length (consists of at least one character).
test *string1* = *string2*	Exit status is 0 (true) if the strings are equal.
test *string1* != *string2*	Exit status is 0 (true) if the strings are not equal.

- Combined tests. The available operators are:

Table 6-7

Option	Description
test ! *condition*	Exit status is 0 (true) if the condition is not true
test *condition1* -a *condition2*	Exit status is 0 (true) if both conditions are true
test *condition1* -o *condition2*	Exit status is 0 (true) if either condition is true

For more detailed information about test, enter **help test** or **man test** (the built-in test command and the external one have identical features).

Use the tr Command

The tr command translates (replaces) or deletes characters. It reads from standard input and prints the result on standard output. With tr, you can replace regular characters or sequences of such characters and special characters like \t (horizontal tab) or \r (return).

A complete list of all special characters handled by tr is included in the man page of the program.

The following is the standard syntax of tr:

tr *set1 set2*

The characters included in ***set1*** are replaced with the characters included in **set2**.

The following is an example of using the tr command:

cat text-file | tr a-z A-Z

This a command causes all lowercase characters in a file to be changed to uppercase, and the result is printed to stdout.

You can use tr to delete characters from the first set by entering the following:

tr -d set1

This will not translate anything; it only deletes the ones included in set1, printing the rest to standard output.

The following is another example of using the tr command:

VAR='echo $VAR | tr -d %'

In this example, tr deletes the percent sign from the original value of VAR and the result is assigned as a new value to the same variable.

By entering a command like

tr -s *set1 char*

you can also use tr to replace a set of characters with a single character.

Exercise Answers

The following are answers to the exercises in this section.

Solution for Exercise 6-1:

Script: **/exercise/section_6/hello.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script prints a "Hello world" greeting
# Author: Tux Penguin
# Created: 8/22/2005

echo -e "\aHello\nworld"
exit 0
```

Solution for Exercise 6-2:

Script: **/exercise/section_6/name1.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script reads the users first and last name
# and then prints a greeting with the full name.
# Author: Tux Penguin
# Created: 8/22/2004

echo "Please enter your first name:"

# first name gets assigned to variable FIRSTNAME
read FIRSTNAME

echo "Please enter your last name:"

# last name gets assigned to variable LASTNAME
read LASTNAME

#Now print the greeting:
echo "Welcome to the club, $FIRSTNAME $LASTNAME"
exit 0
```

Solution for Exercise 6-3:

Script: **/exercise/section_6/name2.sh** on your *3038 Course CD*

```
#!/bin/bash
# This scripts reads the users first and last name
# and then prints a greeting with this full name.
# Author: Tux Penguin
# Created: 8/22/2005

echo "Please enter your first name:"

# first name gets assigned to variable FIRSTNAME
read FIRSTNAME

echo "Please enter your last name:"

# last name gets assigned to variable LASTNAME
read LASTNAME

# create a new NAME variable
NAME="$FIRSTNAME $LASTNAME"

# Now print the greeting:
echo "Welcome back home, $NAME"

exit 0
```

Solution for Exercise 6-4:

Script: **/exercise/section_6/info.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script prints information about
# the current login
# and the current working directory.
# Author: Tux Penguin
# Created: 8/22/2005

login=`whoami`
path=`pwd`

echo "The current login is: $login"
echo "The current path is: $path"
exit 0
```

Solution for Exercise 6-5:

This script uses all available methods for arithmetic operations.

Script: **/exercise/section_6/sum.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script lets the user specify two whole
# numbers and then adds them together. All kinds of
# arithmetic formats that are possible
# under Bash are used, one after another.
# Author: Tux Penguin
# Created: 8/22/2005

declare -i INTEGER1
declare -i INTEGER2
declare -i SUM

# read first integer
echo "Please enter first integer: "
read INTEGER1

# read second integer
echo "Please enter second integer: "
read INTEGER2

# this uses  expr  for Bourne shell compatibility:
RESULT=`expr $INTEGER1 + $INTEGER2`
echo "The  expr  command returns the result: $RESULT."

# this uses the Bash built-in  let :
let RESULT="$INTEGER1 + $INTEGER2"
echo "The  let  built-in returns the result: $RESULT."

# this uses a Bash-specific arithmetic expression:
RESULT=$[$INTEGER1 + $INTEGER2]
#or:
#RESULT=$(($INTEGER1 + $INTEGER2))

echo "Using an arithmetic expression in Bash, the result is: $RESULT."

# this one uses the variables declared as integers #above:
SUM=INTEGER1+INTEGER2
echo "Using the variables declared as integers, the sum is: $SUM."

exit 0
```

Solution for Exercise 6-6:

Script: **/exercise/section_6/find.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script searches for files in the current
# directory.
# The user is prompted to enter a filename;
# if no name is entered, we search for the default
# value anyway, which is set to "*.bak"
# Author: Tux Penguin
# Created: 8/22/2005

echo "Please enter the file to be searched for (default is: *.bak):"
read FILE
find . -name "${FILE:="*.bak"}"
exit 0
```

Solution for Exercise 6-7:

Script: **/exercise/section_6/file_check.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script checks whether a file exists and if
# its executable
# Author: Tux Penguin
# Created: 8/22/2005

echo "Please enter a filename: "

read FILENAME

if test -e $FILENAME
then
  if test -x $FILENAME
  then
    echo "The file exists and is executable."
  else
    echo "The file exists but is not executable."
  fi
else
    echo "The file does not exist."
fi

exit 0
```

Solution for Exercise 6-8:

Script: **/exercise/section_6/yes_no.sh** on your *3038 Course CD*

```
case "$VARIABLE" in
  [yY] | [yY][eE][sS] | [yY] [eE] [aA] [hH] )
    ... ;;
  [nN] | [nN][oO] | [nN][oO][pP][eE] )
    ... ;;
  * )
    echo error message ;;
esac
```

Solutions for Exercise 6-9:

Script: **/exercise/section_6/counter1.sh** on your *3038 Course CD*

```
#!/bin/bash
# A script to iterate over a simple "while" loop 100
# times.
# Author: Tux Penguin
# Created: 8/22/2005

declare -i COUNTER=1

while  test $COUNTER -le 100
do
  echo "The counter stands at $COUNTER."
  COUNTER=COUNTER+1
  sleep 1
done

exit 0
```

Script: **/exercise/section_6/counter2.sh** on your *3038 Course CD*

```
#!/bin/bash
# A script to iterate over a simple until loop 100 times.
# Author: Tux Penguin
# Created: 8/22/2005

declare -i COUNTER=1

until   test $COUNTER -gt 100
do
  echo "The counter stands at $COUNTER."
  COUNTER=COUNTER+1
  sleep 1
done

exit 0
```

Solution for Exercise 6-10:

Script: **/exercise/section_6/lowercase1.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script renames all files in the current
# directory so that they have all lowercase file
# names.
# Author: Tux Penguin
# Created: 8/22/2005

for     FILE in `find . -type f -maxdepth 1`
do
  NEWFILE=`echo $FILE | tr [A-Z] [a-z]`
  if test $FILE != $NEWFILE
  then
    echo mv $FILE $NEWFILE
  fi
done

exit 0
```

Solution for Exercise 6-11:

Script: **/exercise/section_6/lowercase2.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script renames all files in the current
# directory so that they have all-lowercase file
# names.
# 2nd version: Now we also check whether the file
# already exists with lowercase lettering.
# Author: Tux Penguin
# Created: 8/22/2005

for FILE in `find . -type f -maxdepth 1`
do
  NEWFILE=`echo $FILE | tr [A-Z] [a-z]`
  if test $FILE != $NEWFILE
  then
    if test -e $NEWFILE
    then
      echo "There is already a file
            with the name $NEWFILE."
      echo "$FILE will not be renamed."

# Skip the rest and begin next loop iteration:
      continue
    fi
    echo mv $FILE $NEWFILE
  fi
done
exit 0
```

Solution for Exercise 6-12:

For testing purposes, an echo is put before all important commands, such as chown and userdel. There should be no spaces between [yY][eE][sS]. The same is true of [nN][oO].

Script: **/exercise/section_6/userdel1.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script prompts for a user name and
# then deletes the corresponding account.
# Author: Tux Penguin
# Created: 8/22/2005
yesno (){
  while   true
  do
    echo "$*"
    echo "Please answer by entering (y)es or (n)o:"
    read ANSWER
    case "$ANSWER" in
      [yY] | [yY][eE][sS] )
        return 0
        ;;
      [nN] | [nN][oO] )
        return 1
        ;;
      * )
        echo "I can't understand you over here."
        ;;
    esac
  done
}
read -p "Delete which user? " user

if yesno "Also delete home directory of $user?"
then
  home=yes
fi

if yesno "Really delete user $user?"
then
  if test "$home" = yes
  then
    userdel -r $user
  else
    home="/home/$user"
    chown -R root.root $home
    userdel $user
  fi
fi
exit 0
```

Solution for Exercise 6-13:

Script: **/exercise/section_6/userdel2.sh** on your *3038 Course CD*

```
#!/bin/bash
# This script prompts for a user name and then deletes
# the corresponding account. Optionally, the user's
# home directory is deleted as well.
# Author: Tux Penguin
# Created: 8/22/2005

while getopts u:r variable
do
  case $variable in
    u ) user="$OPTARG" ;;
    r ) home=yes ;;
  esac
done

if test "$home" = yes
  then
    userdel -r $user
  else
    home="/home/$user"
    chown -R root.root $home
    userdel $user
fi

exit 0
```

Summary

Objective	Summary

1. Use Basic Script Elements	■ Before writing a shell script, it is useful to draw a program flow chart.
	■ Before a file can be run as a shell script, it must have both read and execute permissions.
	■ To produce some simple output from a script, you can use the echo command.
	■ To read user input for processing by a script, you can use the read command.
	■ There are several ways to perform arithmetic operations in a script:
	  ■ Use the external command expr.
	  ■ Use the Bash built-in command let.
	  ■ Enclose arithmetic expressions in double parentheses for expansion by the shell.
	  ■ In Bash, arithmetic operations can also be performed with plain variables, provided that these have been declared as integers before.

2. Use Variable Substitution Operators	■ In Bash, you can use special variable substitution operators to assign different values to variables without having to rely on external commands. ■ These special substitution operators allow changing variables by deleting certain patterns in their values and returning the rest, for instance. ■ They also allow you to set a default for a variable for situations where no value can be assigned to it.
3. Use Control Structures	■ Conditional statements in shell scripts can be implemented with an `if` branch. ■ For relatively simple structures, you can also use the command separators && and || to express the same statement as a command line. ■ To take decisions with a number of possible choices in a script, create a multiple branch with a `case` statement. ■ With the commands `while` and `until`, create loops that depend on certain terminating conditions. ■ The `for` command allows you to create loops to process a list of elements.

3. Use Control Structures (continued)	■ There are 2 ways to influence the operation of a loop:
	  ■ With the break command, a loop can be terminated completely according to a given condition.
	  ■ The continue command allows exiting from the current iteration of a loop if the condition is true.

Objective	Summary
4. Use Advanced Scripting Techniques	■ If you anticipate that certain command sequences will be used more than once in a script or if you want to make a complex script easier to read and understand, consider defining shell functions for certain routines. ■ A function normally comprises a part of a script and makes it available under a user-definable name, such that the script part can be executed simply by stating this name further below in the script. ■ Use the Bash built-in command getopts to easily extract command-line options for shell scripts. ■ With the getopts command, you can tell the script which options it should recognize and which action should be triggered by a given option.

5. Learn About Useful Commands in Shell Scripts	■ You can use external commands in Shell scripts to perform certain tasks.
	■ The following is a list of commonly-used commands:
	  ■ cat
	  ■ cut
	  ■ date
	  ■ echo
	  ■ grep and egrep
	  ■ sed
	  ■ test
	  ■ tr