Sh Programming 10

Abstract

This chapter covers sh programming. It explains sh scripts and different versions of sh. It compares sh scripts with C programs and points out the difference between interpreted and compiled languages. It shows how to write sh scripts in detail. These include sh variables, sh statements, sh built-in commands, regular system commands and command substitution. Then it explains sh control statements, which include test conditions, for loop, while loop, do-until loop, case statements, and it demonstrates their usage by examples. It shows how to write sh functions and invoke sh functions with parameters. It also shows the wide range of applications of sh scripts by examples. These include the installation, initialization and administration of the Linux system.

The programming project is for the reader to write a sh scripts which recursively copies files and directories. The project is organized in a hierarchy of three sh functions; cpf2f() which copies file to file, cpf2d() which copies file into a directory and cpd2d() which recursively copies directories.

10.1 sh Scripts

A sh script (Bourne 1982; Forouzan and Gilberg 2003) is a text file containing sh statements for the command interpreter sh to execute. As an example, we may create a text file, mysh, containing

```
#! /bin/bash
# comment line
echo hello
```

Use **chmod +x mysh** to make it executable. Then run mysh. The first line of a sh script usually begins with the combination of #!, which is commonly called a **shebang.** When the main sh sees the shebang, it reads the program name for which the script is intended and invokes that program. There are many different versions of sh, e.g. bash of Linux, csh of BSD Unix and ksh of IBM AIX, etc. All the sh programs perform essentially the same task but their scripts differ slightly in syntax. The shebang allows the main sh to invoke the proper version of sh to execute the script. If the shebang is not specified, it runs the default sh, which is /bin/bash in Linux. When bash executes the mysh script, it will print hello.

10.2 sh Scripts vs. C Programs

sh scripts and C programs have some similarities but they are fundamentally different. The following lists a sh script and a C program side by side in order to compare their syntax form and usage.

```
------------ sh ----------------------- C ------------------
INTERPRETER: read & execute |   COMPILE-LINKED to a.out
                            |
 mysh a  b  c  d            |   a.out a b c d
  $0  $1 $2 $3 $4           |   main(int argc, char *argv[ ])
-------------------------------------------------------------
```

First, sh is an **interpreter**, which reads a sh script file line by line and executes the lines directly. If a line is an executable command, sh executes the command directly if it's a built-in command. Otherwise, it forks a child process to execute the command and waits for the child to terminate before continuing, exactly as it does a single command line. In contrast, a C program must be compile-linked to a binary executable first. Then run the binary executable by a child process of the main sh. Second, in C programs, every variable must have a type, e.g. char, int, float, derived types like structs, etc. In contrast, in sh scripts, everything is string. So there is no need for types since there is only one type, namely strings. Third, every C program must have a main() function, and each function must define a return value type and arguments, if any. In contrast, sh scripts do not need a main function. In a sh script, the first executable statement is the entry point of the program.

10.3 Command-line parameters

A sh script can be invoked with parameters exactly as running a sh command, as in

```
mysh one two three
```

Inside the sh script, the command line parameters can be accessed by the position parameters $0, $1, $2, etc. The first 10 command line parameters can be accessed as $0 to $9. Other parameters must be referenced as ${10} to ${n} for n>=10. Alternatively, they can be brought into view by the shift command, which will be shown later. As usual, $0 is the program name itself and $1 to $n are parameters to the program. In sh the built-in variables $# and $* can be used to count and display the command-line parameters.

$# = the number of command-line parameters $1 to $n
$* = ALL command-line parameters, including $0

In addition, sh also has the following built-in variables related to command executions.

$$ = PID of the process executing the sh
$? = last command execution exit status (0 if success, nonzero otherwise)

Example Assume the following mysh script is run as

```
        mysh abc  D  E  F  G  H  I  J  K  L   M  N
           #  1   2  3  4  5  6  7  8  9  10 11 12
1. #! /bin/bash
2. echo \$# = $#            #  $# = 12
3. echo \$* = $*            #  $* = abc D E F G H I J K L M N
4. echo $1  $9  $10         #  abc K abc0  (note: $10 becomes abc0)
5. echo $1  $9  ${10}       #  abc K L       (note: ${10} is L)
6. shift                    # replace $1, $2 .. with $2, $3,...
7. echo $1  $9  ${10}       # D L M
```

In sh, the special char $ means substitution. In order to use $ as is, it must be either single quoted or back quoted by \, similar to \n , \r, \t, etc. in C. In Lines 2 and 3, each \$ is printed as is with no substitution. In Line 4, $1 and $9 are printed correctly but **$10** is printed as **abc0**. This is because sh treats $10 as $1 concatenated with 0. Before echoing, it substitutes $1 with abc, making $10 as abc0. In Line 5, ${10} is printed correctly as L. The reader may try to echo other position parameters ${11} and ${12}, etc. Line 6 shifts the position parameters once to the left, making $2=$1, $3=$2, etc. After a shift, $9 becomes L and ${10} becomes M.

10.4 Sh Variables

Sh has many built-in variables, such as PATH, HOME, TERM, etc. In addition to built-in variables, the user may use any symbol as sh variable. No declaration is necessary. All sh variable values are strings. An unassigned sh variable is the NULL string. A sh variable can be set or assigned a value by

```
      variable=string      # NOTE: no white spaces allowed between tokens
```

If A is a variable, then $A is its value.

Examples

```
   echo A    ==>    A
   echo $A   ==>    (null if variable A is not set)
   A="this is fun"  # set A value
   echo $A   ==>    this is fun
   B=A              # assign "A" to B
   echo $B   ==>    A (B was assigned the string "A")
   B=$A             (B takes the VALUE of A)
   echo $B   ==>    this is fun
```

10.5 Quotes in sh

Sh has many special chars, such as $, /, *, >, <, etc. To use them as ordinary chars, use \ or single quotes to quote them.

Examples:

```
A=xyz
echo \$A       ==> $A       # back quote $ as is
echo '$A'      ==> $A       # NO substitution within SINGLE quotes
echo "see $A" ==> see xyz  # substitute $A in DOUBLE quotes
```

In general, \ is used to quote single chars. Single quotes are used to quote long strings. No substitution occurs within single quotes. Double quotes are used to preserve white spaces in double quoted strings but substitution will occur within double quotes.

10.6 sh Statements

sh statements include all Unix/Linux commands, with possible I/O redirections.

Examples:

```
ls
ls > outfile
date
cp f1 f2
mkdir newdir
cat < filename
```

In addition, the sh programming language also supports statements for testing conditions, loops and cases, etc. which controls the executions of sh programs.

10.7 sh Commands

10.7.1 Built-in Commands

sh has many built-in commands, which are executed by sh without forking a new process. The following lists some of the commonly used built-in sh commands.

. file	**:**	read and execute file
break [n]	**:**	exit from the nearest nth nested loop
cd [dirname]	**:**	change directory
continue [n]	**:**	restart the nearest nth nested loop
eval [arg . . .**]**	**:**	evaluate args once and let sh execute the resulting command(s).
exec [arg . . .**]**	**:**	execute command by this sh, which will exit
exit [n]	**:**	cause sh to exit with exit status n
export [var ..]	**:**	export variables to subsequently executed commands
read [var . . . **]**	**:**	read a line from stdin and assign values to variables
set [arg . . . **]**	**:**	set variables in execution environment
shift	**:**	rename position parameters $2 $3 . . . as $1 $2. . . **.**
trap [arg] [n]	**:**	execute arg on receiving signal n
umask [ddd]	**:**	set umask to octal ddd
wait [pid]	**:**	wait for process pid, if pid is not given, wait for all active children

The read Command: When sh executes the read command, it waits for an input line from stdin. It divides the input line into tokens, which are assigned to the listed variables. A common usage of read is to allow the user to interact with the executing sh, as shown by the following example.

```
echo -n "enter yes or no : "  # wait for user input line from stdin
read ANS                      # sh reads a line from stdin
echo $ANS                     # display the input string
```

After getting the input, sh may test the input string to decide what to do next.

10.7.2 Linux Commands

sh can execute all Linux commands. Among these, some commands have become almost an integral part of sh because they are used extensively in sh scripts. The following lists and explains some of these commands.

The echo Command: echo simply echoes the parameter strings as lines to stdout. It usually condenses adjacent white spaces into a single space unless quoted.

Examples

```
echo This  is     a   line      # display This is a line
echo "This is     a   line"     # display This is    a   line
echo -n hi                      # display hi without NEWLINE
echo   there                    # display hithere
```

The expr Command: Since all sh variables are strings, we can not change them as numerical value directly. For example,

```
I=123       # I assigned the string "123"
I=I + 1     # I assigned the string "I + 1"
```

Rather than incrementing the numerical value of I by 1, the last statement merely changes I to the string "I + 1", which is definitely not what we hoped (I=124). Changing the (numerical) values of sh variables can be done indirectly by the expr command. Expr is a program, which runs as

```
expr string1 OP string2     # OP = any binary operator on numbers
```

It first converts the two parameter strings to numbers, performs the (binary) operation OP on the numbers and then converts the resulting number back to a string. Thus,

```
I=123
I=$(expr $I + 1)
```

changes I from "123" to "124". Similarly, expr can also be used to perform other arithmetic operations on sh variables whose values are strings of digits.

The pipe Command: Pipes are used very often in sh scripts to act as filters.

Examples:

```
ps -ax | grep httpd
cat file | grep word
```

Utility Commands: In addition to the above Linux commands, sh also uses many other utility programs as commands. These include

awk:	data manipulation program.
cmp:	compare two files
comm:	select lines common to two sorted files
grep:	match patterns in a set of files
diff:	find differences between two files
join :	compare two files by joining records with identical keys
sed:	stream or line editor
sort:	sort or merge files
tail:	print the lasst n lines of a file
tr:	one-to-one char translation
uniq:	remove successive duplicated lines from a file

10.8 Command Substitution

In sh, $A is substituted with the value of A. Likewise, when sh sees `cmd` (in grave quotes) or $(cmd), it executes the cmd first and substitutes $(cmd) with the result string of the execution.

Examples

```
echo $(date)      # display the result string of date command
echo $(ls dir)    # display the result string of ls dir command
```

Command substitution is a very powerful mechanism. We shall show its usage throughout the latter part of this chapter.

10.9 Sh Control Statements

Sh is a programming language. It supports many execution control statements, which are similar to those in C.

10.9.1 if-else-fi statement

The syntax of if-else-fi statement is

```
if [ condition ]   # NOTE: must have white space between tokens
   then
      statements
   else            # as usual, the else part is optional
      statements
fi                 # each if must end with a matching fi
```

Each statement must be on a separate line. However, sh allows multiple statements on the same line if they are separated by semi-colon ;. In practice, if-else-fi statements are often written as

```
if [ condition ]; then
    statements
 else
    statements
fi
```

Examples: By default, all values in sh are strings, so they can be compared as strings by

```
if  [ s1 = s2 ]          # NOTE: white spaces needed between tokens
if  [ s1 != s2 ]
if  [ s1 \< s2 ]         # \< because < is a special char
if  [ s1 \> s2 ] etc.    # \> because > is a special char
```

In the above statements, the left bracket symbol [is actually a **test** program, which is executed as

```
test string1 COMP string2  OR  [ string1 COMP string2 ]
```

It compares the two parameter strings to decide whether the condition is true. It is important to note that in sh **0 is TRUE** and **nonzero is FALSE**, which are exactly opposite of C. This is because when sh executes a command, it gets the exit status of the command execution, which is 0 for success and nonzero otherwise. Since [is a program, its exit status is 0 if the execution succeeds, i.e. the tested condition is true, and nonzero if the tested condition is false. Alternatively, the user may also use the sh built-in variable **$?** to test the exit status of the last command execution.

In contrast, the operators -eq, -ne, -lt, -gt , etc. compare parameters as integer numbers.

```
if [ "123" = "0123" ]      is false since they differ as strings
if [ "123" -eq "0123" ]    is true since they have the same numerical value
```

In addition to comparing strings or numerical values, the TEST program can also test file types and file attributes, which are often needed in file operations.

```
if [ -e name ]     # test whether file name exists
if [ -f name ]     # test whether name is a (REG) file
if [ -d name ]     # test whether name is a DIR
if [ -r name ]     # test whether name is readable; similarly for -w, -x, etc.
if [ f1 -ef f2 ]   # test whether f1, f2 are the SAME file
```

Exercise: How does the test program tests files type and file attributes, e.g. readable, writeable, etc.? In particular, how does it determine whether f1 –ef f2?

HINT: stat system call.

Compound if-elif-else-fi statement: This is similar to if-else if-else in C except sh uses elif instead of else if. The syntax of compound if-elif-else-fi statement is

```
if [ condition1 ]; then
     commands
  elif [ condition2 ]; then
     commands
  # additional elif [ condition3 ]; then etc.
  else
     commands
fi
```

Compound Conditions: As in C, sh allows using **&&** (AND) and || (OR) in compound conditions, but the syntax is more rigid than C. The conditions must be enclosed in a pair of matched double brackets, [[and]].

Examples

```
if [[ condition1 && condition2 ]]; then
if [[ condition1 && condition2 || condition3 ]]; then
```

As in C, compound conditions can be grouped by ( ) to enforce their evaluation orders.

```
if [[ expression1 && (expression2 || expression3) ]]; then
```

10.9.2 for Statement

The for statement in sh behaves as for loop in C.

```
for VARIABLE in string1 string2 .... stringn
  do
    commands
  done
```

On each iteration, the VARIABLE takes on a parameter string value and execute the commands between the keywords do and done.

Examples:

```
for FRUIT in apple orange banana cherry
  do
     echo $FRUIT         # print lines of apple orange banana cherry
  done

for NAME in $*
  do
     echo $NAME          # list all command-line parameter strings
     if [ -f $NAME ]; then
        echo $NAME is a file
     elif [ -d $NAME ]; then
          echo $NAME is a DIR
     fi
  done
```

10.9.3 while Statement

The sh while statement is similar to while loop in C

```
while [ condition ]
  do
    commands
  done
```

sh will repeatedly execute the commands inside the do-done keywords while the condition is true. It is expected that the condition will change so that the loop will exit at some point in time.

Example: The following code segment creates directories dir0, dir1,.., dir10000

```
I=0                     # set I to "0" (STRING)
while [ $I != 10000 ]   # compare as strings; OR  while [ $I \< 1000 ] as numbers
  do
    echo $I             # echo current $I value
    mkdir dir$I         # make directories dir0, dir1, etc
    I=$(expr $I + 1)    # use expr to inc I (value) by 1
  done
```

10.9.4 until-do Statement

This is similar to the do-until statement in C.

```
until [ $ANS = "give up" ]
  do
       echo -n "enter your answer : "
       read ANS
  done
```

10.9.5 case Statement

This is similar to the case statement in C also, but it is rarely used in sh programming.

```
case $variable in
     pattern1)   commands;;  # note the double semicolons ;;
     pattern2)   command;;
     patternN)   command;;
esac
```

10.9.6 continue and break Statements

As in C, **continue** restarts the next iteration of the nearest loop and **break** exits the nearest loop. They work exactly the same as in C

10.10 I/O Redirection

When entering a sh command, we may instruct sh to redirect I/O to files other than the default stdin, stdout, sterr. I/O redirections have the following form and meaning:

> file	stdout goes to file, which will be created if non-existing.
>> file	stdout append to file
< file	use file as stdin; file must exist and have r permission.
<< word	take inputs from "here" file until a line containing only “word"

10.11 Here Documents

An output command can be instructed to take inputs from stdin, echo them to stdout until a prearranged keyword is encountered**.**

Examples:

```
echo << END
   # keep enter and echo lines until a line with only
END
cat << DONE
   # keep enter and echo lines until
DONE
```

These are commonly known as **here documents**. They are typically used in a sh script to generate long blocks of descriptive text without echoing each line individually.

10.12 sh Functions

sh functions are defined as

```
func()
{
  # function code
}
```

Since sh executes commands line by line, all functions in a sh script must be defined **before** any executable statements. Unlike C, there is no way to declare function prototypes in sh scripts. Sh functions are invoked in exactly the same way as executing a sh script file. The sh statement

```
func s1 s2 ... sn
```

calls the sh func, passing as parameters (strings) s1 to sn. Inside the called function, the parameters are referenced as $0, $1 to $n. As usual, $0 is the function name, and $1 to $n are position parameters corresponding to the command-line parameters. When a function execution finishes, $? is its exit status, which is 0 for success and nonzero otherwise. The value of $? can be changed by the explicit return value of a function. However, in order to test $? of the last execution, it must be assigned to a variable and test that variable instead.

Examples of sh functions:

```
#! /bin/bash
testFile()    # test whether $1 is a REG file
{
  if [ -f $1 ]; then
     echo $1 is a REG file
  else
     echo $1 is NOT a REG file
}
testDir()     # test whether $1 is a DIR
{
  if [ -d $1 ]; then
     echo $1 is a DIR
  else
     echo $1 is NOT a DIR
}
echo entry point here # entry point of the program
for A in $*       # for A in command-line parameter list
  do
   testFile $A    # call testFile on each command-line param
   testDir  $A    # call testDir  on each command-line param
  done
```

Exercise: In the following sh program

```
testFile()   # test whether $1 is a REG file; return 0 if yes, 1 if not
{
 if [ -f $1 ]; then
    return 0
 else
    return 1
}
for A in f1 D2       # assume f1 is a REG file, D2 is a DIRectory
 do
    testFile $A      # testFile return $?=0 or 1
    if [ $? = 0 ]; then
       echo $A is a REG file
    else
       echo $A is not a REG file
    fi
 done
```

The result is always "$A is a REG file" even if $A is a directory. Explain why?

Modify the program code to make it work properly.

10.13 Wild Cards in sh

Star Wildcard: The most useful wildcard in sh is the *, which expands to all files in the current directory.

Examples

file * :	list information of all files in the current directory.
ls *.c :	list all files ends with .c in the current directory.

? Wildcard: inquires chars in a file name

Examples

file ??? :	all files names with exactly 3 chars
ls *.?? :	all file names with 2 chars following a dot.

[ ] Wildcard : Inquire about chars within a pair of [ ] in filenames.

Examples

file *[ab]* :	all file names containing the chars a or b
ls *[xyz]* :	list all file names containing x or y or z
ls *[a-m]* :	list all file names containing chars in the range of a to m

10.14 Command Grouping

In sh scripts, commands can be grouped together by either { } or ( ).

{ ls; mkdir abc; ls; } : execute the list of commands in { } by the current sh. The only usefulness of command grouping by { } is to execute them in the same environment, e.g. to redirect I/O for all the commands in the group.

The more useful command grouping is by () which is executed by a subsh (process).

(cd newdir; ls; A=value; mkdir $A): execute commands in ( ) by a subsh process. The subsh process may change its working directory without affecting the parent sh. Also, any assigned variables in the subsh will have no effect when the subsh process terminates.

10.15 eval Statement

```
eval [arg1 arg1 .. argn]
```

eval is a built-in command of sh. It is executed by sh itself without forking a new process. It concatenates the input argument strings into a single string, evaluates it once, i.e. perform variable and command substitutions, and present the resulting string for sh to execute.

Example:

```
 a="cat big.c | more"
$a      # error: because sh would execute the command cat with big.c, |, more as
                 files. cat would fail on | because it's not a file.
 eval $a # OK   : eval substitutes $a with "cat big.c | more" first, then let sh
                 execute the resulting command line cat big.c | more.
```

Example: Assume

```
A='$B';  B='abc*'; C=newdir; CWD=/root; /root/newdir is a DIR
```

For the command line

```
cp $A `pwd`/$C   # grave quoted command pwd
```

sh evaluates the command line in the following steps before executing it.

(1). **Parameter substation:** Scan the command line, substitute any $x with its value but do it only once, i.e. no substitutions again for any resulting $ symbols. The line becomes

```
cp $B `pwd`/newdir
```

(2). **Command substitution:** perform `pwd` with substitution. Sh will execute the resulting line

```
cp $B /root/newdir
```

This will result in an error if the current directory does not have any file named $B. However, if we change the original command line to

```
eval cp $A $(pwd)/$C
```

sh will evaluate the command line first before executing it. After eval, the line becomes

```
cp abc* /root/newdir
```

(3). **Wildcard expansion:** When sh executes the line, it expands **new*** to file names that begin with **abc**. This will copy all file names beginning with **abc** to the target directory.

It should be noted that we can always achieve the same effect of eval manually with a few additional statements. Using eval saves a few substitution statements but it may also makes the code hard to understand. Therefore, any unnecessary use of eval should be avoided.

10.16 Debugging sh Scripts

A sh script can be run by a sub sh with the –x option for debugging, as in

```
bash -x mysh
```

The sub sh will show each sh command to be executed, including variable and command substitutions, before executing the command. It allows the user to trace command executions. In case of error, sh will stop at the error line and show the error message.

10.17 Applications of sh scripts

sh scripts are most often used to perform routine work that involves lengthy sequence of commands. We cite some examples where sh scripts are found to be very useful.

Example 1: Most users have the experience of installing Linux to their computers. The Linux installing packages are written in sh scripts. During the installing process, it can interact with the user, find out available hard disks on the users computer, partition and format the disk partitions, download or extract files from the installation media and install the files to their directories until the installation process completes. Attempting to install Linux manually would be very tedious and almost impossible.

Example 2: When a user login to a Linux system, the login process executes a series of sh scripts,

```
.login,  .profile,  .bashrc, etc.
```

to configure the execution environment of the user process automatically. Again, it would be impractical to do all these steps manually.

Example 3: Instead of using Makefiles, simple compile-link tasks can be done by sh scripts containing the compile and link commands. The following shows a sh script which generates a MTX operating system kernel image and user mode command programs on a virtual disk named VFD (Wang 2015).

```
#---------- mk script of the MTX OS -------------
VFD=mtximage
# generate MTX kernel image file
as86 -o ts.o ts.s          # assemble .s file
bcc  -c -ansi t.c          # compile .c files
ld86 -d -o mtx ts.o t.o OBJ/*.o mtxlib 2> /dev/null
# write MTX kernel image to /boot directory of VFD
mount -o loop $VFD /mnt
cp mtx /mnt/boot           # mtx kernel image in /boot directory
umount /mnt
# generate User mode command binaries in /bin of VFD
(cd USER; mkallu)          # command grouping
echo all done
#---------------- end
```

Example 4: Create user accounts for a CS class on a Linux machine. At the author's institution, enrollment in the CS360, Systems Programming, class averages 70 students every semester. It would be too tedious to create all the students accounts on a Linux machine by hand. This is done by the following steps using sh scripts.

(1). The Linux command

```
sudo useradd -m -k DIR -p PASSWORD -s /bin/bash LOGIN
```

creates a new user account with login name LOGIN and password PASSWORD. It creates a user home directory /home/LOGIN, which is populated with files in the -k DIR directory. It creates a line in /etc/passwd for the new user account and adds a line in /etc/shadow for the encrypted PASSWORD. However, the PASSWORD used in the useradd command must be encrypted password, not the original password string. Converting password string to encrypted form can be done by the **crypt** library function. The following C program enc.c also converts a PASSWORD string into encrypted form.

```
/******* enc.c file *********/
#define _XOPEN_SOURCE
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[ ])
{
  printf("%s\n", crypt(argv[1], "$1$"));
}
# gcc -o enc -lcrypt enc.c   # generate the enc file
```

(2). Assume that roster.txt is a class roster file containing lines of students ID and name

```
ID name                # name string in lowercase
```

(3). Run the following sh script **mkuser < roster.txt**

```
# --------------- mkuser sh script file --------------------
#! /bin/bash
while read LINE        # read a line of roster file
do
  I=0
  for N in $LINE       # extract ID as NAME0, name as NAME1
  do
    eval NAME$I=$N     # set NAME0, NAME1, etc.
    I=$(expr $I + 1)   # inc I by 1
  done
  echo $NAME0 $NAME1   # echo ID and name
  useradd -m -k h -p $(./enc $NAME0) -s /bin/bash $NAME1
done
```

(4). User home DIR contents: Every user's home directory, /home/name, can be populated with a set of default DIRs and files. For the CS360 class, every user home directory is populated with a public_html directory containing an index.html file as the user's initial Web page for the user to practice Web programming later.

10.18 Programming Project: Recursive File Copy by sh Script

The programming project is for the reader to use most sh facilities to write a meaningful and useful program in sh script. The requirement is to write a sh script

```
            myrcp f1 f. .... .  fn-1 fn
```

which copies f1, f2, .. fn-1 to fn, where each fi may be either a REG/LNK file or a DIR. For simplicity, exclude special files, which are I/O devices. The myrcp program should behave exactly the same as Linux's command **cp –r** command. The following is a brief analysis of the various cases of the program requirements.

(1). n<2: show usage and exit
(2). n>2: fn must be an existing DIR.
(3). n=2: copy file to file or file to DIR or DIR1 to DIR2
(4). Never copy a file or DIR to itself. Also, do not copy if DIR2 is a descendant of DIR1

The above case analysis is incomplete. The reader must complete the case analysis and formulate an algorithm before attempting to write any code.

Hint and Helps: for the command line **myrcp f1 f. . . . . fn**

```
(1). echo n = $#                            # n = value of n
(2). last = \S{$#}; eval echo last = $last  # last = fn
(3). for NAME in $*
     do
       echo $NAME                           # show f1 to fn
     done
(4). Implement the following functions
     cpf2f(): copy $1 to $2 as files, handle symbolic links
     cpf2d(): copy file $1 into an existing DIR $2 as $2/$(basename $1)
     cod2d(): recursively copy DIR $1 to DIR $2
```

```
# Sample solution of myrcp program

cpf2f()     # called as cpf2f  f1  f2
{
   if [ ! -e $1 ]; then
      echo no such source file $1
      return 1
   fi
   if [ $1 -ef  $2 ]; then
      echo "never copy a file to itself"
      return 1
   fi
  if [ -L $1 ]; then
     echo "copying symlink $1"
     link=$(readlink $1)
       ln -s $link $2
       return 0
   fi
   echo "copying $1 to $2"
   cp $1 $2  2> /dev/null
}
cpf2d()     # called as cpf2d file DIR
{
   newfile=$2/$(basename $1)
   cpf2f $1 $newfile
}
cpd2d()     # called as cpd2d dir1 dir2
{
   # reader FINISH cpd2d code
}
# **************** entry point of myrcp ****************
# case analysis;
# for each f1 to fn-1 call cpf2f() or cpf2d() or cpd2d()
```

10.19 Summary

This chapter covers sh programming. It explains sh scripts and different versions of sh. It compares sh scripts with C programs and points out the difference between interpreted and compiled languages. It shows how to write sh scripts in detail. These include sh variables, sh statements, sh built-in commands, regular system commands and command substitution. Then it explains sh control statements, which include test conditions, for loop, while loop, do-until loop, case statements, and it demonstrates their usage by examples. It shows how to write sh functions and invoke sh functions with parameters. It also shows the wide range of applications of sh scripts by examples. These include the installation, initialization and administration of the Linux system.

The programming project is for the reader to write a sh scripts which recursively copies files and directories. The project is organized in a hierarchy of three sh functions; cpf2f() which copies file to file, cpf2d() which copies file into a directory and cpd2d() which recursively copies directories.

References

Bourne, S.R., The Unix System, Addison-Wesley, 1982
Forouzan, B.A., Gilberg, R.F., Unix and Shell Programming, Brooks/Cole, 2003
Wang, K.C., Design and Implementation of the MTX Operating System, Springer International Publishing AG, 2015