Programming the UNIX/linux Shell

Claude Cantin (claude.cantin@nrc.ca)
http://www.nrc.ca/imsb/rcsg

Research Computing Support Group
Information Management Services Branch
National Research Council

April 16, 2006

This page intentionally left blank.

Claude Cantin
National Research Council of Canada

History of printing:

Date	Copies
March 2003	200
March 2001	200
June 1999	200
November 1997	200
July 1996	200
November 1995	150
March 1995	150
February 1994	150
October 1993	100
August 1993	75
February 1993	75
November 1992	35
September 1992	40
February 1992	50
December 1991	50
April 1991	50
September 1990	40
January 1990	40

Table 0.1: Printings.

Contents

List of Tables

Chapter 1

Programming the Shell

The basics of both the `Bourne` and the `C` shells have already been introduced. This chapter will concentrate on programming the shells.

All examples shown will use the `Bourne` shell. Most shell scripts are written in the `Bourne` shell because it is the only shell found on *ALL* UNIX systems. Scripts written in the `Korn` shell are gaining popularity.

`C` shell scripts are possible, but not recommended. Although the `C` shell is great for interactive work, it has many drawbacks in script programming.

1.1 Introduction

1.1.1 `Bourne -- C` shell comparisons

A `Bourne Shell Script` is a file containing a series of `Bourne Shell` commands, as well as control structures such as `if` statements, `while` loops, etc. Parameters can be passed to the script and data can be read from the keyboard.

Bourne shell scripts are usually Algol-66 look-alikes, whereas C shell scripts look more like C program constructs.

The C shell is slower to start execution (as it looks in the `.cshrc` file EVERY TIME it is called) and produces a hash table of all `paths` for faster execution. The Bourne shell is slower, but starts executing immediately.

For that reason, short scripts are executed much faster in the Bourne Shell than the C shell (this is noticeable for small SLOW machines; with systems running at 30 to 130 MIPS/CPU, the difference in speed is almost insignificant). Large scripts should most likely

be written in a compiled language—preferably C—because compiled code executes much faster than interpreted code.

Many features of the `Bourne` shell are also found in the `C` shell. Parameters are interpreted in the same way. Variable assignments are very similar. UNIX commands are exactly the same. Control structures and comparisons differ the most.

More information on the C shell can be found in [7, pp. 484-494] and your systems manuals. The Bourne shell is discussed in [7, pp. 415-461] and your systems manuals.

Again, all examples shown in this chapter will follow the `Bourne` shell syntax.

1.1.2 Variables

Variables are, by the author's convention, always in UPPER CASE CHARACTERS. This makes them easier to recognize.

Variables are assigned a value(s) using an assignment operation:

`VARIABLE`=*string*

Note that there are NO spaces before and after the equal (=) sign: this is essential, otherwise the shell would interpret those spaces.

The content of the variable is represented by prefixing the variable with a dollar sign ($). For example, `$FRUIT` means the content of variable `FRUIT`.

1.2 A Simple Script

```
#!/bin/sh
echo "Hello World."
exit 0
```

The first line of every script should start with a hash sign followed by an exclamation mark. This means "use the program that follows to run this script". In this example, the program used to run the above script is `/bin/sh`, or the `Bourne` shell. The program could have been `/bin/csh` for the `C` shell, or `/bin/ksh` for the `Korn` shell, or any other program that could interpret the commands that follow.

The first line may also contain flags for the program. Examples of such flags are

`-v` This causes every line to be displayed as it is executed.

`-x` Prints the commands and their parameters as they are executed. Useful for debugging.

`-f` For the `C` shell. Inhibit execution of `.cshrc`.

The second line of the program simply displays the string `Hello World.`

REMEMBER: every script should start with `#!`*shell_name* where *shell_name* is the name of the shell in which the script is written.

NOTE: the `echo` command may include special characters:

\n: newline.
\c: suppress newline.
\t: insert TAB.

When such special characters are used, some shells, namely `bash` in linux, may require the `-e` flag. Others do not require it. Experimentation will tell you if you need that flag or not.

1.2.1 Running the Script

To run the above script, one may issue

sh *program*

Or, after giving the program execute permission (**chmod +x** *program*), just

program

1.3 Special Characters

As with the `Bourne` shell features already covered, scripts:

- can contain filenames and directory names with metacharacters.
- can use semicolons (`;`) to separate commands on one line.
- can use redirection signs ($<, >, >>$).
- understand pipes (|).

In addition to the above-mentioned, scripts may also contain `C` and `Bourne` shell features.

1.3.1 '...': Turn off Meaning of Special Characters.

Single quotes are used within the shell and within the scripts to turn off the meaning of special characters, including spaces. A dollar sign ($) will normally be interpreted as a dollar sign, not the content of the variable following that $ sign.

For example,

```
sh_prompt> VARI=me
sh_prompt> echo $VARI
sh_prompt > me
sh_prompt> echo '$VARI'
sh_prompt> $VARI
```

1.3.2 "...": Turn off Meaning of Special Characters EXCEPT $ and ‘

Within double quotes, all special characters are interpreted as their ASCII characters (not what they represent). This excludes the dollar sign ($) and the back quote (‘).

Adding to the above example,

```
sh_prompt> echo VARI = $VARI
VARI = me
sh_prompt> echo "VARI = $VARI"
VARI = me
```

NOTE the spaces in the above example: they are printed as seen (spaces are significant within quotes).

1.3.3 ‘...‘: Use Output as Content of Variable

To give a variable the value of the output of a command, one uses back quotes:

```
sh_prompt> VARI="me"
sh_prompt> echo $VARI
me
sh_prompt> OTHER=‘echo $VARI‘
sh_prompt> echo $OTHER
me
```

The output of the command within the back quotes is taken to be the new content of the first variable. Another short example:

```
sh_prompt> TODAY=‘date‘
sh_prompt> echo $TODAY
Mon Aug 20 17:35:51 EDT 1990
sh_prompt> echo ”today’s date is $TODAY”
today’s date is Mon Aug 20 17:35:51 EDT 1990
```

Note that by default, the output of a series of commands is sent to the standard output. If the output is preceded by an = sign, the variable preceding the equal sign takes the value of the output of the command.

1.4 Verbatim echo

Contents of variables and strings can easily be displayed with the `echo` command. `cat` may be used to display multi-line strings without using multiple `echo` commands.

cat << ’*string*’
This is part of the script being output in a verbatim fashion. All lines preceding ‘string’ would be displayed as seen in the script itself.
string

The message to be displayed must be terminated with *string* on a line by itself, with no leading blanks.

Verbatim echo is useful when lengthy instructions are required for the user of a shell script to help him/her proceed.

1.5 Parameters and the Shell

A shell is invoked by typing its name. Parameters are passed to the script by appending them to the script name, with spaces as separators.

1.5.1 $0: The Name of the Invoking Command

The special variable $0 represents the name of the executing program. The following shell, if called script.sh would output This program is called script.sh.:

```
#!/bin/sh
echo This program is called $0.
exit 0
```

1.5.2 $1 $2 $3 ... $9, $*: Shell Parameters

The first parameter to the shell is known as $1, the second as $2, etc. The collection of ALL parameters is known as $*.

Consider the following as an example (file prog):

```
#!/bin/sh
echo the first parameter is $1
echo the second parameter is $2
echo the collection of ALL parameters is $*
exit 0
```

The output of that program could be:

```
sh_prompt> prog first second
the first parameter is first
the second parameter is second
the collection of ALL parameters is first second
sh_prompt>
```

Only nine parameters can be read using the $*number* scheme. The $* along with the shift instruction is one way to read the remainder of the parameters. Another way is to use the for *var* in $* command, to be described in the upcoming Control Structures section.

1.5.3 $#: Number of Parameters

The number of parameters used can be obtained by looking at the value of $#.

1.5.4 shift: Shifts Parameters

When a large number of parameters (> 9) are passed to the shell, `shift` can be used to read those parameters. If the number of parameters to be read is known, say three, a program similar to the following could be written:

```
#!/bin/sh
echo The first parameter is $1.
shift
echo The second parameter is $1.
shift
echo The third parameter is $1.
exit 0
```

Obviously the above example contains redundancy, especially if there are a large number of parameters.

To solve this problem: use a `for` or `while` loop.

1.6 read: Reading Input from User

The following short example shows how `read` can be used to get input from the user:

```
#!/bin/sh
echo -e "Please enter your name: \c"
read NAME
echo "Your name is $NAME."
exit 0
```

The \c means that the line feed will be suppressed, so that the prompt sits at the end of the line, not at the beginning of the following line.

Two more common controls available to the `echo` command are to use \n to add a line feed, and \t to add a tab.

Multiple values may be read on a single line by using:

```
#!/bin/sh
echo -e "Please enter two numbers: \c"
read NUM1 NUM2
echo The numbers entered are $NUM1 and $NUM2
exit 0
```

This ensures that if two numbers are entered on a single line, they will be read within two variables. If three numbers were entered, the second variable (NUM2) would contain the last two numbers.

Assuming three numbers were the input of the above example, the first two numbers could be assigned to the first variable by entering them as

num1\ num2 num3

The backslash (\) allows the blank space between *num1* and *num2* to be part of the variable (ordinarily, spaces are used as `field seperators`.

1.7 `test`: Comparisons

A common requirement of many programs is to compare two, three, or more things together. Strings and numbers may be compared. Files are often checked for their lengths and/or existence.

All such verifications are done using variants of the `test` command.

The general usage of `test` is

`test` *expression*

If *expression* is true, a return code of 0 is supplied.

If *expression* is false, a non-zero return code is generated.

1.7.1 Testing/Comparing Numbers

The primitives available for comparison of numeric values are

- `-eq`, `-ne`: equal, not equal.
- `-gt`, `-lt`: greater, less than.
- `-ge`, `-le`: greater or equal, less or equal.

For example:

```
#!/bin/sh

if test $# -le 5
```

```
then
    echo Less than or equal to five parameters.
else
    echo More than 5 parameters.
fi
exit 0
```

1.7.2 Verifying File Types

To test file types, a number of primitives are used (taken from [7, p. 439]):

`-s` checks that the file exists and is not empty.

`-f` checks that the file is an ordinary file (not a directory).

`-d` checks whether the file is really a directory.

`-x` checks that the file is executable.

`-w` checks that the file is writeable.

`-r` checks that the file is readable.

An example would be where a program needs to output something to a file, but first checks that the file exists:

```
#!/bin/sh
if test ! -s arg.file
then
    echo "arg.file is empty or does not exist."
    ls -l > arg.file
    exit
else
    echo "File arg.file already exists."
fi
exit 0
```

Note the exclamation mark within the `test` sequence. The exclamation mark means “not”.

1.7.3 Comparing Strings

String comparisons are done using = and `!=`:

```
#!/bin/sh
if test $# -eq 0
then
    echo Must provide parameters.
    exit 1
fi

while test ! $1 = "end"
do
    echo parameter is $1
    shift
    if test $# -eq 0
    then
        echo Parameter list MUST contain the '"'end'"' string.
        exit
    fi
done
echo Done: I"'"ve hit the '"'end'"' string.
exit 0
```

Note that the above example could have been MUCH shorter if no error checking took place.

The length of strings can also be tested using:

`-z` : check if the string has zero length.

`-n` : check if the string has a non-zero length.

1.7.4 Combining `test`: Expressions

Two or more `test` expressions may be combined, using the `-o` (or) and/or the `-a` (and) attributes:

```
#!/bin/sh
if test $# -eq 0
then
```

```
    echo Must provide parameters.
    exit 1
fi

if test $# -gt 2 -a $# -lt 5
then
    echo There are 3 or 4 parameters.
fi

if test $# -ge 1 -a $# -lt 3
then
    echo There are 1 or 2 parameters.
fi
exit 0
```

Note that `-a` has precedence over `-o`.

1.8 Control Structures

1.8.1 `for` *var* `in` *list*

The last example could easily be rewritten as

```
#!/bin/sh
for i in $*
do
    echo The parameter is $i.
done
exit 0
```

In the above program `$*` could have been replaced by any list of names. The program would then be changed to:

```
#!/bin/sh
for i in Claude Paul Wayne Roger Tom
do
    echo The name is $i.
done
exit 0
```

Within the shell, parameters are read as `$1` for the first parameter, `$2` for the second parameter, `$3` for the third parameter, and so on. `$*` is the entire list of parameters.

If the "`in` *list*" is omitted, the list taken is the list of parameters passed to the shell on the command line.

`seq`: sequence of numbers (linux)

If the script is expecting to go through a sequence of numbers, the `seq` command may be used to control the looping:

```
#!/bin/sh
for i in ‘seq 5 15‘
do
    echo "Number in sequence is $i."
done
exit 0
```

`seq` is a linux command that prints a sequence of numbers. There are 3 ways to use it: with one, two or three parameters:

- 1 parameter: sequence starts at one, increment of one, and ends at *num1*.
- 2 parameters: sequence starts at *num1*, increment of one, and ends at *num2*.
- 3 parameters: sequence starts at *num1*, increment of *num2*, and ends at *num3*

num1, *num2* and *num3* are the first, second, and third parameters to `seq`. Numbers may be fractions and negative numbers are allowed.

`seq` is a linux-specific command. Neither IRIX nor Solaris include that command.

1.8.2 `while` *condition*

The *condition* involves the `test` statement seen above, where one value is compared with another. If the comparison is true, the commands within the loop are executed. If not, they are not executed.

Using another form of the above example,

```
#!/bin/sh
while test $# -gt 0; do
```

```
    echo parameter $1
    shift
done
exit 0
```

Note that the `while` and `do` are on the same line, and that in this case they must be separated by a semicolon. The semicolon is a command separator.

1.8.3 `until` loop

Again, another way to write the above example is to use the `until` control sequence:

```
#!/bin/sh
until test $# -eq 0
do
    echo parameter $1
    shift
done
exit 0
```

1.8.4 `if` statement

What would happen if no parameters had been passed to the script? Here is how one can verify that there is at least one parameter in the list:

```
#!/bin/sh
if test $# -eq 0
then
    echo There must be at least one parameter on the command line
else
    until test $# -eq 0
    do
        echo parameter $1
        shift
    done
fi
exit 0
```

Another way to write this same program is

```
#!/bin/sh
if test $# -eq 0
then
    echo There must be at least one parameter on the command line
    exit 1
fi
until test $# -eq 0
do
    echo parameter $1
    shift
done
exit 0
```

Note that the format of the `if` statement is

`if` *comparison*
`then`
. . .
`else`
. . .
`fi`

The `else` is optional, or if followed by another `if`, `else if` could be replaced by `elif`. The statement is ended by `fi` ("if" spelled backwards).

1.8.5 `case`: Selections

Often, one wants to perform specific actions on specific values of a variable. This is an example of such a program:

```
#!/bin/sh
case $# in
    0) echo no parameters;;
    1) echo only one parameter.
       echo put commands to be executed here.;;
    *) echo more than one parameter.
       echo enter code here.;;
esac
exit 0
```

Wildcards can be used as items within the `case` statement. Hence `*` means "anything matches".

Note the use of the double semicolons. These are always required to terminate `case` pattern statements.

A much more powerful `case` example would be:

```
while :
do
    echo -e "Would you like to continue? \c"
    read ANS
    case $ANS in
        [yY] | [yY][eE][sS])  echo "Fine, then we'll continue."
            break
            ;;
        [nN] | [nN][oO]) echo "We shall now stop."
            exit
            ;;
        *) echo "You must enter a yes or no verdict!"
    esac
done
echo "\nWe are now out of the while loop."
```

The example shows a number of tricks sometimes used during shell programming. The first is the use of an infinite `while` loop by using the `:` operator instead of a `test` operator. The `:` always returns a successful result (represented by a return code of 0). (In general, the `test` operator is actually not needed – but is almost always used – to control loops. As `test` always returns `0` for a successful comparison and `1` for a non-successful comparison, it is easy to use. In reality, any appropriate UNIX command could be used to control the loops. The loop would then be entered if a return code of `0` were returned by the command.)

The second trick is done with the use of the `break` command. When `break` is encountered, the execution flow branches to the end of the current loop. In this case, the `while` loop.

Finally, the content of `ANS` is compared with a number of *pattern matching sequences*. Those pattern matching sequences actually replace a large number of `if` statements. In our case, the statement

```
[Yy] | [Yy][Ee][Ss] )
```

would match `Y`, `y`, `YES`, `YEs`, `Yes`, `yES`, `yEs`, `yeS`, `YeS` and `yes`. The vertical bar `|` is an `OR` sign.

In fact, all shell metacharacters are allowed in forming a pattern match string. In the example above:

```
[a-z]* )
```

would match any string beginning with a lower case character.

1.9 `expr`: Doing Arithmetic

`expr` is used to perform arithmetic manipulations. Five functions can be used:

`+` addition.

`-` subtraction.

`*` multiplication.

`/` division.

`%` remainder.

Here is an example that uses all of them:

```
#!/bin/sh
if test $# -lt 2 -o $# -gt 2
then
    echo Must provide two and only two parameters.
    exit 1
fi
SUM=‘expr $1 + $2‘
DIFF=‘expr $1 - $2‘
PRODUCT=‘expr $1 "*" $2‘
QUOTIENT=‘expr $1 / $2‘
REM=‘expr $1 % $2‘
TOTAL=‘expr $SUM + $DIFF + $PRODUCT + $QUOTIENT + $REM‘
echo The sum is $SUM.
echo The difference is $DIFF.
```

```
echo The product is $PRODUCT.
echo The quotient is $QUOTIENT.
echo The remainder is $REM.
echo The total sum of all these numbers is $TOTAL.
exit 0
```

NOTES:

- The entire `expr` line is enclosed within back quotes. Normally, the output of `expr` is the standard output. Including the back quotes will assign the result to a variable.
- The `*` is between double quotes. If it was not, the shell would interpret it as the wildcard character.
- `expr` is a shell feature found in both the `Bourne` and `C` shells.

1.10 Subroutines

Scripts, like any other programming language, may contain subroutines. A common way to use a subroutine is:

```
#!/bin/sh

usage()
{
    echo " "
    echo "You have not used this program correctly."
    echo "Use as:  prog_name par1 par2 ..."
    echo " "
    exit 1
}

one_par()
{
    echo
    echo There was only one parameter on the command line.
    echo
    return 1
```

```
}

two_par()
{
    echo
    echo There were two parameters on the command line.
    echo
    return 1
}

larger()
{
    echo
    echo There were many parameters.
    echo
    return 1
}

case $# in
    0) usage;;
    1) one_par ;;
    2) two_par ;;
    *) larger ;;
esac
exit 0
```

Note that routines must be defined prior to being used and that all variables are global.

1.11 Miscellaneous

1.11.1 `break`

This command tells the shell to exit the innermost loop it is currently in.

1.11.2 $$: Process ID

Within a script, $$ represents the Process ID of the currently running script. This allows unique file names for temporary files created by the script. For example

```
/tmp/temp.$$
```

is assured a unique file name.

1.11.3 $!: PID (Process ID) of the previous background command

1.11.4 $?: Return Code

Any command always returns a Return Code, represented by the $? variable. A successful command usually returns 0.

1.11.5 A || B: Either or

A and B each represent a distinct set of commands. If the return code from set A is 0 (successful), then the B side will not be executed.

If the return code from set A is non-zero (unsuccessful), then set B will be executed.

For example,

```
#!/bin/sh
FILE="/usr/tmp/junk"
test -f $FILE || {
    echo "File $FILE does not exist."
    echo "Let's create it."
    touch $FILE
}
```

1.11.6 A && B: A or Both

A and B each represent a distinct set of commands. If the return code from set A is 0 (successful), set B will be executed. Otherwise, set B will not be executed.

For example,

```
#!/bin/sh
FILE="/usr/tmp/junk"
test -f $FILE  && rm $FILE
```

first verifies that the file exists, then erases it. If it did not exist, the `rm` command would not have been issued.

A series of commands could have been executed, as shown in the previous "`||`" example.

1.11.7 `set`: Change Command-line Parameters

Typically, command line parameters are typed on the same line as the command. Those parameters may be replaced at any time using the `set` command:

```
#!/bin/sh
echo "Current parameters are $*."
set ‘date +%B‘
echo "The current month is $1."
```

If there were no parameters on the command line, `set` would create the parameters.

`set` is sometimes used to generate separate fields out of a pathname. A trick is to use it along with redefining `IFS`, the variable that controls what a field separator is:

```
set -- ‘(IFS=/; echo $HOME)‘
```

This command temporarily defines "`/`" as a field separator, and sets each directory as a separate argument to the command line. The "`--`" ensures that if the variable being displayed (in this case `$HOME`, contains a dash (`-`), that character would be interpreted as a dash, not as an option to the command.

NOTE: this works only with the `bash` shell.

1.12 `trap`: Trap signals

Processes may be sent signals using either the `kill` command, or a control key combination such as `CTRL-C`. The *interrupt* signal (`CTRL-C`) usually kills the process.

Table 1.1 taken from [6, p. 883] shows some of the signals used in Shell Programming.

The `trap` command typically appears as one of the first lines in the shell script. It contains the commands to be executed when a signal is detected as well as what signals to trap.

```
#!/bin/sh
TMPFILE=/usr/tmp/junk.$$
```

Signal Number	Signal Name	Explanation
0	Normal exit	`exit` command.
1	SIGHUP	When session disconnected.
2	SIGINT	Interrupt – often CTRL-c.
15	SIGTERM	From `kill` command.

Table 1.1: Signal numbers for `trap` command.

```
trap 'rm -f $TMPFILE; exit 0' 1 2 15
.
.
.
```

Upon receiving signals 1, 2 or 15, `$TMPFILE` would be deleted and the script would terminate the shell script normally. This shows how `trap` may be used to clean up before exiting.

```
#!/bin/sh
TMPFILE=/usr/tmp/junk.$$
trap '' 0 1 2
.
.
.
```

The above example shows how `trap` may be used to ignore specific signals (0, 1 and 2), while NOT interrupting the current line of execution.

NOTE that when the signal is received, the command currently being executed is interrupted (except in the case where there is nothing to be executed between the two quotation signs), and execution flow continues at the next line of the script.

1.13 Tips Using Shell Commands

This section discusses a few UNIX commands sometimes used in scripts.

1.13.1 basename: Current Directory

`basename` *directory*

returns the portion of *directory* after the last "/", i.e. the current directory name.

For example,

```
#!/bin/sh
CUR_DIR=‘basename \$cwd‘
```

will set `CUR_DIR` to be the current directory name from the last "/" to the end (if the current directory is `/usr/people/cantin`, it would return `cantin`).

1.13.2 dirname: Directory Path

`dirname` *directory*

will return the path of the current directory from the beginning of the directory name to the last "/".

For example,

```
dirname /usr/people/cantin
```

will return `/usr/people`. But

```
dirname cantin
```

will return `.` (dot).

For example, the following script will take as its first parameter, the name of a directory, and return its absolute path name. The directory must be given either as a relative or absolute path name.

```
#!/bin/sh
# Returns full path name of a directory.
if [ $# -ne 1 ]
then
    echo " "
    echo " $0: must have one parameter."
    echo " "
    exit
```

```
fi

C_DIR=‘pwd‘
if test "‘dirname $1‘" = "."
then
    FULL_NAME=$C_DIR/$1
else
    FULL_NAME=$1
fi
echo "Absolute path name is $FULL_NAME."
```

1.13.3 `stty -echo`: Passwords

If the script requires the input of a “secret” string, the string typed from the keyboard should not be displayed on the screen. To stop the display of the characters typed,

```
stty -echo
```

may be used.

To resume echoing of the characters,

```
stty echo
```

is used.

```
#!/bin/sh
echo -e "Please enter your passcode: \c"
stty -echo
read PASSCODE
stty echo
echo "The passcode was not seen when typed."
```

1.13.4 `cut`, `awk`: Cut Selected Fields

`cut` and `awk` may both be used to select a specific field from the output of a command. A typical application would be to create a list of currently logged-in users:

```
#!/bin/sh
#
# using cut:
USERS=‘who | cut -f1 -d" "‘
echo "Users on the system are: $USERS."

# using awk:
USERS=‘who | awk ’{print $1}’‘
echo "Users on the system are: $USERS."
```

In the first example, `cut` takes its input from the `who` command. `-f1` specifies that the first field is to be cut out from each line, and `-d" "` specifies that the field delimiter is a space.

In the second example, `awk` takes its input the same way `cut` did, then prints the first field (space is the default field delimiter for `awk`).

In general, `cut` is a much simpler command to use. `awk` is a full language; scripts can be written using `awk` as the language instead of the shell script.

1.14 Exercises

1. I have an IBM REXX emulator on my UNIX machine. I also have a set of REXX files I want to run. What should I do to those files in order to get them to run simply by typing their file name? Assume the REXX emulator is `/usr/local/bin/rexx`.

2. Write a script that would display the time when run.

3. Write a script that would wait 5 seconds, then display the time, but 5 seconds late. Hint: use the ”sleep” command.

4. Write a script that will take a person’s name as a parameter to the program name. The script should greet that person, as

   `Good Day` *name_entered*`, How are you today`?

5. Write a script that will prompt the user for a name. That same name will then be displayed in the same format as the previous exercise.

6. Write a script that will return the number of parameters on the command line.

7. Write a series of scripts that will `count` the number of parameters on the command line, first using the `for` statement, then the `while` and finally the `until` statement. (Three scripts).

8. Write a script that will echo the third parameter, but only if it is present.

9. Given a file of numbers (one per line), write a script that will find the lowest and highest number.

10. Write a script that would recognize if a word entered from the keyboard started with an upper or lower case character or a digit. Use the "case" statement.

    The script would then output the word, followed by "upper case", "lower case", "digit", or "not upper, lower, or digit".

11. Write a script that would first verify if file "myfile" exists. If if does not, create it, then ask the user for confirmation to erase it...

12. Write a `Bourne` shell script to automatically compile your Fortran or C programs (*prog.f* or *prog.c*). The script should, if a name is not provided in the command line, prompt the user to input the program to be compiled. Hint: use `awk -F. 'print $1'`.

13. Write a program that will calculate the amount of disk space your directory is using on the system.

    Hint: the program could use both the `du` and `awk` commands.

14. Write a program that will list the files you have in the current directory, followed by the directories. The output should have the form:

```
Files:
------
filenames

Directories:
------------
directories
```

15. Write a program to verify how many users are logged on to the system.

16. Write a program that will take an undetermined list of parameters, and reverse them.

17. Write a script that will accept any number of parameters. The program should display whether an odd or an even number of parameters was given.

18. Write a script asking the user to input some numbers. The script should stop asking for numbers when the number 0 is entered. The output should look like:

```
user: logon_name

Lowest number entered:
Highest number entered:
Difference between the two:
Product of the two:
```

19. Write a shell script that uses a temporary file to store the user name and the time the script starts. The script should run for approximately 30 seconds (use the "sleep" command). Run another occurrence of that same script, ensuring the temporary file used really has a unique name.

    The script should verify that the temporary file does not exist prior to attempting to use it.

20. Write a script that changes the command line parameters to your own login id.

21. Write a script that ignores any "CTRL-C". In fact, when "CTRL-C" is entered, it should display a message, then continue to run.

22. Write a script that reads a "password" from a user (it will not show when typed, but will get displayed after the carriage return is entered).

Chapter 2

Solutions to Exercises

This section represents possible solutions to the exercises in the course material.

2.1 Programming the Shells

The scripts shown below are all Bourne shell based. C shell scripts could also be written to solve these problems.

Most of the scripts lack error-checking mechanisms such as verifying the number of arguments, checking if a file exists before creating it, etc. The addition of such tests is good programming practice; their inclusion is left to the reader as further exercises.

1. Once the `REXX` script is written, the execution mode should be set (`chmod +x` *script.rexx*). NOTE, the first line of the script must look like

   ```
   #!/usr/local/bin/rexx
   ```

   so that the `rexx` shell is invoked to run the commands.

2. This exercise makes use of the grave accents (also called back quotes) to get the output from the `date` command.

   ```
   #!/bin/sh
   echo "The current date is ‘date‘."
   exit 0
   ```

3. The output from the `date` command is stored in a variable and redisplayed 5 seconds later.

```
#!/bin/sh
DATE=‘date‘
echo "Current: $DATE."
sleep 5
echo "5 seconds late: $DATE."
exit 0
```

4. The argument on the program line is referred to as `$1`.

```
#!/bin/sh

if test $# -eq 0
then
    echo "No name on command line."
    exit 1
fi

echo "Good Day $1, How are you today?"
exit 0
```

5. The only difference with the previous exercise is the way the name is read by the script:

```
#!/bin/sh

echo "Enter you name here: "; read NAME

echo "Good Day $NAME, How are you today?"
exit 0
```

6. This script is really a “one-liner”:

```
#!/bin/sh

echo "There were $# arguments on the command line."
exit 0
```

7. This solution is:

```
#!/bin/sh

NUM=0
for i in $*
do
    NUM=‘expr $NUM + 1‘
done
echo "There were $NUM arguments."
exit 0
```

```
#!/bin/sh

NUM=0
while test $# -gt 0
do
    NUM=‘expr $NUM + 1‘
     shift
done
echo "There were $NUM arguments."
exit 0
```

```
#!/bin/sh

NUM=0
until test $# -eq 0
do
```

```
    NUM=‘expr $NUM + 1‘
     shift
done
echo "There were $NUM arguments."
exit 0
```

8. First, one should verify that the third parameter actually exists:

```
#!/bin/sh

if test $# -ge 3
then
     echo "The third argument is $3."
else
     echo "There were only $# parameters."
     echo "Program stopped."
fi
exit 0
```

9. One such script could be

```
#!/bin/sh
# script to print lowest and highest number from a file.
#
if test $# -eq 0
then
     echo usage: minmax filename
     exit 1
fi

sort -n numbers > sorted.numbers

read SMALLEST < sorted.numbers
LARGEST=‘tail -1 sorted.numbers‘
```

```
rm sorted.numbers

echo " "
echo "The smallest number is $SMALLEST."
echo "The largest number is $LARGEST."
echo " "
  exit 0
```

This script should contain code to check that `sorted.numbers` does not already exist. It is left to the reader to add that code.

10. This makes use of pattern matching within the `case` statement.

```
#!/bin/sh
echo -e "Enter a word or number: \c"; read ANS
case $ANS in
  [a-z]*) echo "lower case"
        ;;
  [A-Z]*) echo "upper case"
        ;;
  [0-9]*) echo "number."
        ;;
  *) echo "not upper, lower, or number."
esac
exit 0
```

11. This uses an infinite `while` loop and the `break` command.

```
#!/bin/sh
FILE="myfile"
if test -f $FILE
then
   echo "File already exist."
else
   echo "creating myfile."
```

```
        touch myfile
        while :
        do
            echo -e "would you like to erase it? \c"
            read ANS
            case $ANS in
               [yY] | [yY][eE][sS])  echo "Fine, then we'll erase it."
                        rm myfile
                        break
                        ;;
               [nN] | [nN][oO]) echo "OK we will keep it, then."
                        break
                        ;;
               *) echo "You must enter a yes or no!"
            esac
        done

    fi
    exit 0
```

12. This is a toughy, especially if you have no idea what `awk` is. The following script will take either a FORTRAN or a C program, and compile it using the default flags:

```
#!/bin/sh

if test $# -eq 0
then
    echo " "
    echo "Enter program to be compiled."
     read PROGRAM
else
    PROGRAM=$1
fi

NAME=‘echo $PROGRAM | awk -F. '{print $1}'‘
make $NAME
```

13. This script uses `du` and `awk` to find the solution:

```
#!/bin/sh

CURRENT_DIR=‘echo $cwd‘
cd
cd ..
LAST_LINE=‘du $user | tail -1‘
SPACE=‘echo $LAST_LINE | awk ’{print $1}’‘

echo "My home directory is using $SPACE kilobytes."

cd $CURRENT_DIR
```

14. This script makes use of the creation of a few temporary files, and of the `read` statement on one of those files. Note where the input redirection sign is placed: this ensures that the file is kept open during the read process.

```
#!/bin/sh

ls > /tmp/listing
SIZE=‘cat /tmp/listing | wc -l‘
NUM=0
while [ $NUM -lt $SIZE ]
do
     read FILE
     if [ -f $FILE ]
     then
         echo $FILE >> files
    elif [ -d $FILE ]
    then
        echo $FILE >> dirs
    fi
    NUM=‘expr $NUM + 1‘
```

```
done < /tmp/listing

echo "Files:"
echo "-----"
cat files
echo " "
echo "Directories:"
echo "-----------"
cat dirs

rm files dirs /tmp/listing
```

Note that no `test` strings appear in this program. Instead, open and closed square brackets were used (`[ ]`). These can be used interchangeably instead of the `test` command.

15. This is simple:

```
#!/bin/sh

echo "There are ‘who | wc -l‘ users in the system."
```

16. Again, this is fairly straightforward:

```
#!/bin/sh

if [ $# -eq 0 ]
then
     echo There were no arguments on the command line.
     exit
fi

ARGS=""
while [ !  $# -eq 0 ]
do
```

```
    ARGS="$1 $ARGS"
     shift
done

echo "The reversed arguments are $ARGS"
```

17. This script makes use of the % operation:

```
#!/bin/sh

REM=‘expr $# % 2‘
if [ $REM -eq 0 ]
then
    echo "There were an even number of arguments."
else
    echo "There were an odd number of arguments."
fi
exit 0
```

18. Again, this script makes use of the `expr` command.

```
#!/bin/sh

HIGHEST=0
LOWEST=99999999
echo "Please input numbers: "
read NUMBER
while [ $NUMBER -ne 0 ]
do
    if test $NUMBER -gt $HIGHEST
     then
         HIGHEST=$NUMBER
     fi
     if test $NUMBER -lt $LOWEST
     then
```

```
        LOWEST=$NUMBER
     fi
    read NUMBER
done
echo " "
  echo "user: $USER"
echo " "
if [ $HIGHEST -eq 0 -a  $LOWEST -eq 99999999 ]
then
     echo "No numbers were entered."
else
     echo "Lowest number entered: $LOWEST"
     echo "Highest number entered: $HIGHEST"
     DIFF=‘expr $HIGHEST - $LOWEST‘
     echo "Difference between the two: $DIFF"
     PROD=‘expr $LOWEST "*" $HIGHEST‘
     echo "Product of the two: $PROD"
    echo " "
fi
```

19. This could be useful for unique temporary file names.

```
#!/bin/sh
TMPFILE="/tmp/an16.$$"
if test -f $TMPFILE
then
  echo "Temporary file exists!!!"
  exit
fi

date > $TMPFILE
echo $USER >> $TMPFILE

echo -e "\ncontent of $TMPFILE is:"
echo "------"
cat $TMPFILE
```

```
sleep 30
rm $TMPFILE
exit 0
```

Run the script in the background, then start another. Their temporary file names will not conflict.

20. To change command line parameters:

```
#!/bin/sh
echo "Current parameters are: $*"
set $USER
echo "New Current parameters are: $*"
exit 0
```

21. This uses the `trap` command.

```
#!/bin/sh
trap 'echo "Ignoring Control-C..."' 2
for i in 1 2 3 4 5 6 7 8
do
  sleep 2
done
echo "program now terminated normally..."
exit 0
```

22. Useful to hide passwords.

```
#!/bin/sh

echo -e "\nEnter a secret word: \c"
stty -echo
```

```
read SECRET
stty echo

echo -e "\nYour secret word is $SECRET."
exit 0
```

Chapter 3

Bibliography

Bibliography

[1] Stephen R. Bourne. *The UNIX System V environment*. Addison-Wesley Publishing Company. Don Mills, Ontario. 1987.

[2] D. Dougherty, R. Koman, and P. Ferguson. *The Mosaic Handbook for the X Window System*. O'Reilly & Associates, Inc. Sebastopol, California. 1994.

[3] E. Foxley. *UNIX for Super-Users*. Addison-Wesley Publishing Company. Don Mills, Ontario. 1985.

[4] Æleen Frisch. *Essential System Administration*. O'Reilly & Associates, Inc. Sebastopol, California. 1992.

[5] Ed Krol. *The Whole INTERNET User's Guide & Catalogue*. O'Reilly & Associates, Inc. Sebastopol, California. 1994.

[6] Jerry Peek, Tim O'Reilly, and Mike Loukides. *UNIX Power Tools*. O'Reilly & Associates, Inc. Sebastopol, California. 1993.

[7] H. McGilton and R. Morgan. *Introducing the UNIX SYSTEM*. McGraw-Hill Software Series for Computer Professionals. Toronto. 1983.

[8] R. Thomas, and R. Farrow. *UNIX Administration Guide for System V*. Prentice Hall. Englewood Cliffs, New Jersey. 1989.

[9] Silicon Graphics Inc. *IRIS-4D User's Guide*, man pages.

[10] Sun Microsystems. *SunOS 4.0, 4.1 Reference Manuals*.

[11] SuSE Linux LTD. *SuSE Linux 7.3 Reference Manual*.

[12] UNIX International. *The UNIX Operating System: A Commercial Success Story*. Nov 1, 1989. Parsippany, NJ.

[13] http://www.canarie.ca; November 1997 version.