Shell Programming

Author: Reuben Francis Cornel

1 Why program using the shell

- It makes life easy. Batching a whole list of commands and putting them in a file to make a script saves on time as well as the effect remembering all those commands can have on your grey cells.
- Portability. If scripts are written using a very generic notation. Such as notation used by the `sh` shell script. It can be ported to almost any UNIX system readily.

2 Different Shells

- **Bourne SHell** [**sh**]: This was the first shell developed for UNIX . Most of the shells we have today derive their syntax and semantics from the Bourne Shell.
- **C SHell** [**csh**]: This was developed by Bill Joy at Berkeley. This shell is not generally recommended for scripting because of the complexity of its syntax.
- **Korn SHell** [**ksh**]: The Korn Shell can be considered the superset of the Bourne Shell as far as the features provided are concerned. But it is not fully portable as the Bourne Shell, as some systems don't support ksh.
- **Bourne Again SHell** [**bash**]: This is the GNU version of the Bourne shell with some added features. Most Linux systems provide BASH.

3 Initialization Files

- **.profile**: This file is the file which is executed as soon as you login. As a result, most commands used to setup the terminal envirnoment are put in this file.
- **.exrc**: This is the initialisation file for vi

4 In the beginning there was ... The Interpreter Line

- Most shell scripts generally follow a standard pattern, which is shown below.

  Interpreter Line
  commands

- The interpreter line if present, must always be on the first line, and it follows the pattern.

  #!<*path to interpreter*>

- If the interpreter line is not present, then in that case you have to run the script from the command line by specifically mentioning the interpreter.

5 Shell Script Execution

- To execute any shell script, first you have to set its executable permissions. This is done as shown below.

  `chmod +x` *filename*

- To execute the shell script, if you are present in the same directory as the path all you have to do is type *./filename*

6 Variables

6.1 Creating Variables

- Like most computer languages, the shell also provides variables. But unlike most computer languages. The variables are not strongly typed.
- To create a new variable all you have to do is just assign a value to a variable name.A variable name has to start with a alphabet, but can contain numbers later on, spaces are absolutely not allowed. The structure of this statement is shown below.

  <Variable Name>=<Value>

- Please note there is **no** space surrounding the "=".
- To reference the variable you prefix the variable with a $.
- Another point to which you have to pay special attention is the fact that shell variable names are case sensitive. They won't raise an error when you don't declare them, but use them. This again could be a cause for a number of bugs.

6.2 Scope of shell variables

- When a shell variable is assigned a value. Its scope is limited only to the current shell process. Therefore if a sub-shell is started it is oblivious of the value of a variable declared in the parent shell.
- The `export` command is used to increase the visiblity of a shell variable. Therefore its visibility extends to children shells of the current parent shell.
- If a variable declared in the parent shell is modified in the child shell, its changed value is not reflected in the parent shell.
- If we want the change reflected in the parent shell, we have to *source* the script. This is nothing but running the script in our current interactive shell process than in a new shell process. The usage of this feature is shown below.

  ._<script name>

  The underscore represents a space

6.3 Positional Parameters

- There are a set of variables called positional parameters, using which you can access values passed from the command line. These are represented starting from $0...$9. $0 represents the name of the executable. Whereas the variable $1...$9 represent the arguments passed to the command. Ok, so we can access nine parameters but what if we want to access the tenth parameter.
- There are a couple of variables, symbolically `$*` which helps us access more than one variable.
- We have another variable on lines of `$*`, symbolically `$@`. This helps with accessing arguments which have spaces between them.
- Now that we can have access to all arguments, what if we want to access the number of arguments. The variable `$#` lets us do that.
- We have another positional parameter $?. This parameter gets us the return value of the previously executed command. This can be used to check if the previous command has executed properly.

6.4 Default Values

- There are times when we would want a variable to have a default value rather than null. The shell provides us a feature which lets us assign a default value to a variable. Its sytax is as shown below.

```
echo ‘‘${name:=reuben}’’
```

6.5 Environment Variables

- The shell provides us with a set of variables which control our environment.
- The list of most frequently used environment variables is given in the table below.

Variable	Meaning
PATH	This variable is a list of paths which the shell will search for a program to execute.
PS1	This variable represents the text which is displayed at the command prompt
PS2	This variable signifies the text displayed by the secondary command prompt
LOGNAME	This variable stores the user name of the person using the terminal.
IFS	This variable specifies the characters which act as seperators in lists
HOSTNAME	This variable specifies the hostname of the current computer the user has logged onto.

- To get a full listing of variable you can type `env`.

7 Quoting in Shell

- The shell provides us with three types of quotes, which are used depending on the situation.
- Quoting is typically used when you want to pass the entire string to a command.
- **Weak Quoting**: Enclosing a string with *double quotes* is weak quoting. If a string is weak quoted all variables in the string will be substituted with their corresponding values, and other double quotes will have to be "escaped".
- **Strong Quoting**: When we enclose a string in *single quotes*. The shell does not interpret any special charaters or varibles in the string. But it has to be noted that single quotes in the string will have to be escaped.
- **Command Substitution**: There are times when we need the output of a program to be used. In cases like this command substitution comes in helpful. We use *back quotes* for command substitution.

8 Shell Programming Constructs

- Since the shell is quite a bit like a programming language, it provides us with a number of constructs which perform their job well but cannot be compared with their counterparts from full fledged programming languages.
- When using the constructs shown here, please use them verbatim.

8.1 if

- This is one of the most basic control structures provided by the shell. Its syntax is as follows.

```
if command is successful
then
    execute commands
elif command is successful
then
    execute commands
else
    execute commands
fi
```

- The command can be any UNIX program. But generally the program put here is a `test` program. This program serves the purpose which relational operators serve.
- But fortunately, the test command can be abbreviated using [ ]. Therefore, conditions can be as follows [$x -eq $y]

Option	Meaning

continued on the next page

Test Options (*continued*)

Option	Meaning
Numeric Comparision	
-eq	Equal to
-ne	Not Equal
-lt	Less than
-le	Less than and equal to
-gt	Greater Than
-ge	Greater than and equal to
String Comparisions	
s1 == s2	String s1 equals to string s2
s1 != s2	String s1 not equal to string s2
-z s1	String s1 is null
File Operations	
-f	File Exists
-r	File is readable
-w	File is writable
-x	File is executable
-d	File is a directory

8.2 case

- Case statement here provides the multiple branching facility. Its syntax is as given below.

```
case expression in
   patterns ) commands ;;
   patterns ) commands ;;
...
esac
```

- The pattern can contain shell wildcards, such as *, ? etc

8.3 for

- The shell for loop, iterates over a list of elements until all the elements of the list are exhausted. The seperator used is specified by the IFS variable. Its syntax is given below

```
for variable in list
do
    execute commands
done
```

8.4 while

- The while statement is a iterative statement which iterates as long as the condition is true. Like the if statement it uses the return value of the command. This means that you can use the abbreviated form of `test` command here. The syntax is as shown below.

```
while command is successful
do
```

execute commands
`done`

9 Special Commands

- Apart from these contructs the shell also provides a small set of commands which help in control flow.

Command	Syntax	Meaning
trap	`trap` *command signal-list*	This command is used in cases where there is a probability of the script being stopped in the middle, it perform clean up operations specified by command
exit	`exit`	The script exits
break	`break`	Control from a while or a for loop is broken
return	`return`	Used as a return from functions
set	`set` *arguments*	Assigns arguments to positional parameters.

10 Math Operations using the Shell

- There are 2 ways of doing math operations using the shell. All basic math operators are supported by the shell.
  - **expr**: This command evaluates the expression given to it.
  - **$((*expression*))**: This is an abbreviated form, used for math operations.

11 String Operations

- **Concatenation**: To concatenate two strings all you have to do is refer one shell variable after another with out a space
- **Length of String**: We can use the `expr` command to find the length of a given string. The syntax is given below.

```
expr "thbs" :  ".*"
```

- **Extracting a substring**: Again the expr command can be utilised to extract a substring from a given string.

```
expr "thbs" :  '..\(..\)'
```

12 Functions in Shell

- The syntax of shell functions are as shown below.

function_name() {
    statements
    return *value*
}

- Arguments are passed using the positional parameters which can be passed as if calling any other function.
- The return statement takes a *numerical argument only.*

13 Debugging Shell Scripts

- Now that you have completed writing your shell script which you consider a *piece-de-resistance.* You discover there is an irritating bug. Here our old friend `set` comes in helpful. All you have to do is include `set -x` in the script. This command gives you the execution trace of the script.