Writing Scripts

Hadi Otrok

www.encs.concordia.ca/~h_otrok

h_otrok@encs.concordia.ca

COEN 346

Agenda

- Creating Scripts
- Environmental Variables
- Read Command and If Statement
- Example

Programming or Scripting ?

- bash is not only an excellent command line shell, but a scripting language in itself. Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.
- Difference between programming and scripting languages:
  - Programming languages are generally a lot more powerful and a lot faster than scripting languages. Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.
  - A scripting language also starts from source code, but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction. Interpreted programs are generally slower than compiled programs. The main advantage is that you can easily port the source file to any operating system. bash is a scripting language. Other examples of scripting languages are Perl, Lisp, and Tcl.

Writing a script

- Open a text editor; for example:
  - $ vi &

  and type the following inside it:
  - #!/bin/bash — This is a special clue given to the shell indicating what program is used to interpret the script. In this case, it is /bin/bash.
  - # My first script — Comment (not read by the interpreter)
  - echo “Hello World”
- The first line tells Linux to use the bash interpreter to run this script. We call it hello.sh. Then, make the script executable:
  - $ chmod 700 hello.sh
  - $ ls –l

  -rwx------ hello.sh

Writing a Script

- Usually, we type the following to run the script:
  - $ ./hello.sh
- Now, the question is: How we can execute the script without typing **./**
- When you type in the name of a command, the system does not search the entire computer to find where the program is located. That would take a long time.
- The shell maintains a list of directories where executable files (programs) are kept, and just searches the directories in that list.
- So, if we add our directory to the list of directories we can run our script as any command.

Writing a Script

- This list of directories is called your *path.*
- *echo $PATH*
- This will return a colon separated list of directories that will be searched if a specific path name is not given when a command is attempted.
- To execute the program without ./ we have to do the following:
  - By setting PATH=$PATH:. our working directory is included in the search path for commands, and we simply type:
  - $ hello.sh

Variables

- We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell language that will convert variables to numbers for calculations.
- We have no need to declare a variable, just assigning a value to its reference will create it.
- Example
  - ```
    #!/bin/bash
    STR="Hello World!"
    echo $STR
    ```
- Line 2 creates a variable called STR and assigns the string "Hello World!" to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.
    ```

Environmental Variables

- There are two types of variables:
  - Local variables
  - Environmental variables
- Environmental variables are set by the system and can usually be found by using the env command. Environmental variables hold special values. For instance,
  - ```
    $ echo $SHELL
    /bin/bash
    $ echo $PATH
    /usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin
    ```
- Environmental variables are defined in /etc/profile, /etc/profile.d/ and ~/.bash_profile. These files are the initialization files and they are read when bash shell is invoked. When a login shell exits, bash reads ~/.bash_logout
    ```

How to use the read command

- The read command allows you to prompt for input and store it in a variable.
- Example (read.sh)
  - ```
    #!/bin/bash
    echo –n “Enter name of file to delete: ”
    read file
    echo “Type 'y' to remove it, 'n' to change your mind ... ”
    rm –i $file
    echo "That was YOUR decision!"
    ```
- Line 3 creates a variable called file and assigns the input from keyboard to it. Then the value of this variable is retrieved by putting the '$' in at its beginning.
    ```

Read

- Options
  - read –s (does not echo input)
  - read –nN (accepts only N characters of input)
  - read –p “message” (prompts message)
  - read –tT (accepts input for T seconds)
- Example

```
$ read -s -n1 -p "Yes (Y) or not (N)?" answer
Yes (Y) or not (N) ? Y
$ echo $answer
Y
```

Single Quotes versus double quotes

- Basically, variable names are exapnded within double quotes, but not single quotes. If you do not need to refer to variables, single quotes are good to use as the results are more predictable.
- Example:
  - #!/bin/bash
  - echo -n '$USER=' # -n option stops echo from breaking the line
  - echo "$USER"
  - echo "\$USER=$USER" # this does the same thing as the first two lines
- The output looks like this (assuming your username is *Student*)
  $USER=Student
  $USER=Student
- So the double quotes still have a work around. Double quotes are more flexible, but less predictable. Given the choice between single quotes and double quotes, use single quotes.

Command Substituation

- The backquote “`” is different from the single quote “´”. It is used for command substitution: `command`
  - `$ LIST=`ls``
    `$ echo $LIST`
    `hello.sh read.sh`
  - `PS1=“`pwd`>”`

  `/home/rinaldi/didattica/>`
- We can perform the command substitution by means of $(command)
  - `$ LIST=$(ls)`
    `$ echo $LIST`
    `hello.sh read.sh`
  - `rm $( find / -name “*.tmp” )`
  - `ls $( pwd )`
  - `ls $( echo /bin )`

Arithmetic

- The **let** statement can be used to do mathematical functions:
  - ```
    $ let X=10+2*7
    $ echo $X
    24
    $ let Y=X+2*4
    $ echo $Y
    32
    ```
- An arithmetic expression can be evaluated by $[expression] or $((expression))
  - ```
    $ echo $((123+20))
    143
    ```
  - ```
    $ VALORE=$[123+20]
    ```
  - ```
    $ echo $[123*$VALORE]
    1430
    ```
  - ```
    $ echo $[2**3]
    ```
  - ```
    $ echo $[8%3]
    ```

Example

- Example (operations.sh)
  - #!/bin/bash
    echo -n “Enter the first number: ”; read x
    echo -n “Enter the second number: ”; read y
    add=$(($x + $y))
    sub=$(($x - $y))
    mul=$(($x * $y))
    div=$(($x / $y))
    mod=$(($x % $y))
    # print out the answers:
    echo “Sum: $add”
    echo “Difference: $sub”
    echo “Product: $mul”
    echo “Quotient: $div”
    echo “Remainder: $mod”

If statement

- Conditionals let we decide whether to perform an action or not, this decision is taken by evaluating an expression. The most basic form is:
  - if [*expression*];
    then
      statements
    elif [*expression*];
      then
        statements
    else
        statements
    fi
  - the elif (else if) and else sections are optional

If Statement

- An expression can be: String comparison, Numeric comparison, File operators and Logical operators and it is represented by [expression]:
- String Comparisons:
  - = compare if two strings are equal
  - != compare if two strings are not equal
  - -n evaluate if string length is greater than zero
  - -z evaluate if string length is equal to zero
- Examples:
  - [s1 = s2] (true if s1 same as s2, else false)
  - [s1 != s2] (true if s1 not same as s2, else false)
  - [s1] (true if s1 is not empty, else false)
  - [-n s1] (true if s1 has a length greater then 0, else false)
  - [-z s2] (true if s2 has a length of 0, otherwise false)

Example

- ```
  #!/bin/bash # if0.sh
  echo -n "Enter your login name: "
  read name
  if [ "$name" = "$USER" ];
  then
        echo "Hello, $name. How are you today ?"
  else
        echo "You are not $USER, so who are you ?"
  fi
  ```
- ```
  #!/bin/bash # if1.sh
  echo -n "Enter a number 1 < x < 10: "
  read num
  if [ "$num" -lt 10 ];    then
     if [ "$num" -gt 1 ]; then
           echo "$num*$num=$(($num*$num))"
        else
           echo "Wrong insertion !"
        fi
  else
        echo "Wrong insertion !"
  fi
  ```