CSE 15L:
Software Tools and Techniques Laboratory

Summer Session I

http://ieng6.ucsd.edu/~cs15u/index.html

Dr. ILKAY ALTINTAS

TA: ALOK SINGH

Lecture 6

July 21st, 2014

Today's Topics

1. Shell scripting programming constructs
   - Shell variables and operators
   - Logic structures

Shell Programming

- Programming features of the UNIX/LINUX shell:
  - *Shell variables*: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory
  - *Operators*: Shell scripts support many operators, including those for performing mathematical operations
  - *Logic structures*: Shell scripts support sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to another), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives)

Variables

- **Variables are symbolic names that represent values stored in memory**
- **Three different types of variables**
  - Global Variables: **Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USERNAME, and PWD.**

    When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.
  - Local Variables

    Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.
  - Special Variables

    Such as positional parameters $1, $2 …

    $1 refers to the first string after the name of the script file on the command line, e.g., echo $1

    $2 refers to the second string, and so on.

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using '$':

e.g.
```
$ echo $HOME

$ echo $SHELL
```

To see a list of your environment variables:

```
$ printenv
```

or:

```
$ printenv | more
```

Defining Local Variables

- As in any other programming language, variables can be defined and used in shell scripts.

Unlike other programming languages, variables in Shell Scripts are not typed.

- Examples :

```
a=1234   # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1 `   will perform arithmetic so b is 1235 now.
```

Note : +,-,/,*,**, % operators are available.

```
b=abcde  # b is string
b='abcde' # same as above but much safer.
b=abc   def # will not work unless 'quoted'
b='abc def'  # i.e. this will work.
```

IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE =

Referencing variables
--curly bracket

- Having defined a variable, its contents can be referenced by the $ symbol. E.g. `${variable}` or simply `$variable`. When ambiguity exists $variable will not work. Use `${ }` the rigorous form to be on the safe side.
- Examples:

```
a='abc'
b=${a}def # this would not have worked without the{ } as
      #it would try to access a variable named adef

 account='cse15f'
 echo ${account%??}
 echo ${account#??}
```

Variable List/Arrary

- To create lists (array) – round bracket

  $ set Y = (UNL 123 CS251)

- To set a list element – square bracket

  $ set Y[2] = HUSKER

- To view a list element:

  $ echo $Y[2]

- Example:

```
     #!/bin/sh
     a=(1 2 3)
     echo ${a[*]}
     echo ${a[0]}
Results:  1 2 3
     1
```

Positional Parameters

- When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.
  - $0 This variable that contains the name of the script
  - $1, $2, ….. $9 1st, 2nd 3rd command line parameter
  - $# Number of command line parameters
  - $$ process ID of the shell
  - $@ same as $* but as a list one at a time (see for loops later)
  - $? Return code ‘exit code’ of the last command
  - `Shift` command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

Example:

Invoke : `./myscript  one two buckle my shoe`

During the execution of `myscrip`t variables $1 $2 $3 $4 and $5 will contain the values *`one, two, buckle, my, shoe`* respectively.

Variables

-
  ```
  vi myinputs.sh
  #! /bin/sh
  echo Total number of inputs: $#
  echo First input: $1
  echo Second input: $2
  ```

-
  ```
  chmod u+x myinputs.sh
  ```

-
  ```
  myinputs.sh ALTINTAS UCSD CSE
     Total number of inputs: 3
     First input: ALTINTAS
     Second input: UCSD
  ```

Shell Programming

- programming features of the UNIX shell:
  - ■ *Shell variables*
  - ■ *Operators*
  - ■ *Logic structures*

Shell Operators

- The Bash/Bourne/ksh shell operators are divided into three groups:
  - defining and evaluating operators,
  - arithmetic operators, and
  - redirecting and piping operators

Defining and Evaluating

- A shell variable take on the generalized form variable=value (except in the C shell).

```
$ set x=37; echo $x
37
$ unset x; echo $x
x: Undefined variable.
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ set mydir=`pwd`; echo $mydir
```

Pipes & Redirecting

- Piping: An important early development in Unix , a way to pass the output of one tool to the input of another.

  ```
  $ who | wc -l
  ```

  By combining these two tools, giving the wc command the output of who, you can build a new command to list the number of users currently on the system

- Redirecting via angle brackets: Redirecting input and output follows a similar principle to that of piping except that redirects work with files, not commands.

  ```
  tr '[a-z]' '[A-Z]' < $in_file > $out_file
  ```

  The command must come first, the *in_file* is directed in by the less_than sign (<) and the *out_file* is pointed at by the greater_than sign (>).

Arithmetic Operators

- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &&, ||
  - parentheses: (,)
  - precedence is the same as C, Java

Arithmetic Operators

- vi math.sh

```
#!/bin/sh
count=5
count=`expr $count + 1 `
echo $count
```

- chmod u+x math.sh
- math.sh

```
6
```

Arithmetic Operators

- vi real.sh

```
#!/bin/sh
a=5.48
b=10.32
c=`echo “scale=2; $a + $b” |bc`
echo $c
```

- chmod u+x real.sh
- ./real.sh

```
    15.80
```

Arithmetic operations in shell scripts

var++ ,var-- , ++var , --var	post/pre increment/ decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& , |	bitwise AND, OR
&& ||	logical AND, OR

Shell Programming

- programming features of the UNIX shell:
  - ■ *Shell variables*
  - ■ *Operators*
  - ■ *Logic structures*

Shell Operators

- The Bash/Bourne/ksh shell operators are divided into three groups:
  - defining and evaluating operators,
  - arithmetic operators, and
  - redirecting and piping operators

Defining and Evaluating

- A shell variable take on the generalized form variable=value (except in the C shell).

  ```
  $ set x=37; echo $x
  37
  $ unset x; echo $x
  x: Undefined variable.
  ```

- You can set a pathname or a command to a variable or substitute to set the variable.

  ```
  $ set mydir=`pwd`; echo $mydir
  ```

Pipes & Redirecting

- Piping: An important early development in Unix , a way to pass the output of one tool to the input of another.

  ```
  $ who | wc -l
  ```

  By combining these two tools, giving the wc command the output of who, you can build a new command to list the number of users currently on the system

- Redirecting via angle brackets: Redirecting input and output follows a similar principle to that of piping except that redirects work with files, not commands.

  ```
  tr '[a-z]' '[A-Z]' < $in_file > $out_file
  ```

  The command must come first, the *in_file* is directed in by the less_than sign (<) and the *out_file* is pointed at by the greater_than sign (>).

Arithmetic Operators

- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &&, ||
  - parentheses: (,)
  - precedence is the same as C, Java

Arithmetic Operators

- vi math.sh

```
#!/bin/sh
count=5
count=`expr $count + 1 `
echo $count
```

- chmod u+x math.sh
- math.sh

```
6
```

Arithmetic Operators

- vi real.sh

```
#!/bin/sh
a=5.48
b=10.32
c=`echo “scale=2; $a + $b” |bc`
echo $c
```

- chmod u+x real.sh
- ./real.sh

```
    15.80
```

Arithmetic operations in shell scripts

var++ ,var-- , ++var , --var	post/pre increment/decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& , |	bitwise AND, OR
&& ||	logical AND, OR

Shell Programming

- programming features of the UNIX shell:
  - *Shell variables*
  - *Operators*
  - *Logic structures*

Shell Logic Structures

The four basic logic structures needed for program development are:

- Sequential logic: to execute commands in the order in which they appear in the program
- Decision logic: to execute commands only if a certain condition is satisfied
- Looping logic: to repeat a series of commands for a given number of times
- Case logic: to replace "if then/else if/else" statements when making numerous comparisons

Conditional Statements (if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
   execute command
elif this command executes successfully
then
   execute this command
   and execute this command
else
   execute default command
fi
```

However- elif and/or else clause can be omitted.

Examples

SIMPLE EXAMPLE:

```
if date | grep “Fri”
then
 echo “It’s Friday!”
fi
```

FULL EXAMPLE:

```
if   [  “$1”  ==  “Monday”  ]
then
 echo “The typed argument is Monday.”
elif [ “$1” == “Tuesday” ]
 then
 echo “Typed argument is Tuesday”
 else
 echo “Typed argument is neither Monday nor Tuesday”
fi
```

# Note: = or == will both work in the test but == is better for readability.

Tests

String and numeric comparisons used with test or [[   ]] which is an alias for test and also [   ] which is another acceptable syntax

string1 = string2	True if strings are identical
String1 == string2	…ditto….
string1 !=string2	True if strings are not identical
string	Return 0 exit status (=true) if string is not null
-n string	Return 0 exit status (=true) if string is not null
-z string	Return 0 exit status (=true) if string is null

int1 -eq int2	Test identity
int1 -ne int2	Test inequality
int1 -lt int2	Less than
int1 -gt int2	Greater than
int1 -le int2	Less than or equal
int1 -ge int2	Greater than or equal

Combining tests with logical operators || (or) and && (and)

- Syntax: if cond1 && cond2 || cond3 …
- An alternative form is to use a compound statement using the –a and –o keywords, i.e.

  if cond1 –a cond22 –o cond3 …

- Where cond1,2,3 .. Are either commands returning a value or test conditions of the form [ ] or test …
- Examples:

```
if  date | grep “Fri”  &&  `date +’%H’` -gt 17
then
    echo “It’s Friday, it’s home time!!!”
fi

if [ “$a” –lt 0 –o “$a” –gt 100 ]     # note the spaces around ] and [
then
    echo “ limits exceeded”
fi
```

File inquiry operations

-d file	Test if file is a directory
-f file	Test if file is not a directory
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists
-z file	Test if the file has zero length

All these conditions return true if satisfied and false otherwise.

Decision Logic

- A simple example

```
#!/bin/sh
if [ "$#" -ne 2 ] then
       echo $0 needs two parameters!
       echo You are inputting $# parameters.
 else
       par1=$1
       par2=$2
 fi
 echo $par1
 echo $par2
```

Decision Logic

Another example:

```
#! /bin/sh
#  number is positive, zero or negative
echo —e "enter a number:\c"
read number
if [ "$number" -lt 0 ]
then
      echo "negative"
elif [ “$number” -eq 0 ]
then
      echo zero
else
       echo positive
fi
```

Loops

- Loop is a block of code that is repeated a number of times.
- The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (for loops) or until a particular condition is satisfied (while and until loops)
- To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

for loops

- Syntax:

```
 for  arg in list
        do
               command(s)
                    ...
        done
```

- Where the value of the variable *arg* is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.
- Example:

```
for i in 3 2 5 7
do
        echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

The while Loop

- A different pattern for looping is created using the while statement
- The while statement best illustrates how to set up a loop to test repeatedly for a matching condition
- The while loop tests an expression in a manner similar to the if statement
- As long as the statement inside the brackets is true, the statements inside the do and done statements repeat

while loops

- Syntax:

```
while this_command_execute_successfully
 do
     this command
     and this command
 done
```

- EXAMPLE:

```
while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
do
    i=`expr $i - 1`
done
```

Looping Logic

- Example:

```
#!/bin/sh
for person in Bob
  Susan Joe Gerry
do
  echo Hello $person
done
```

Output:

```
Hello Bob
Hello Susan
Hello Joe
Hello Gerry
```

- Adding integers from 1 to 10

```
#!/bin/sh
i=1
sum=0
while [ “$i” -le 10 ]
 do
     echo Adding $i into
the sum.
     sum=`expr $sum + $i`
     i=`expr $i + 1 `
 done
echo The sum is $sum.
```

until loops

- The syntax and usage is almost identical to the while-loops.
- Except that the block is executed until the test condition is satisfied, which is the opposite of the effect of test condition in while loops.

- Note: You can think of *until* as equivalent to *not_while*

Syntax:

```
until test
do
  commands ….
done
```

Switch/Case Logic

- The switch logic structure simplifies the selection of a match when you have a list of choices
- It allows your program to perform one of many actions, depending upon the value of a variable

Case statements

- The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
case argument in
    pattern 1) execute this command
               and this
               and this;;
    pattern 2) execute this command
               and this
               and this;;
    esac
```

Functions

- Functions are a way of grouping together commands so that they can later be executed via a single reference to their name. If the same set of instructions have to be repeated in more than one part of the code, this will save a lot of coding and also reduce possibility of typing errors.

  SYNTAX:

  ```
   functionname()
    {
         block of commands
     }
  ```

```
#!/bin/sh

  sum() {
    x=`expr $1 + $2`
    echo $x
    }


sum 5 3

echo "The sum of 4 and 7 is `sum 4 7`"
```

Next Week

- XML and HTML
- Building with Ant
- More debugging

Conditionals

```
if [ "$var" = "hello world" ]

then

  echo "goodbye world"

fi
```

for Loops

```
for i in {1..10}
do
   if [ $[ i % 2 ] = 1 ]; then
      echo $i "is odd"
   fi
done
```

```
for i in {1..10}
do
    if [ $[ i % 2 ] = 1 ]; then
      echo $i "is odd”
    else
      echo $i "is even”
    fi
done
```

Case

```
for i in {1..10}
do
      case $[ i % 3 ]
        in
           0)
        echo $i "apples”
        ;;
           1)
        echo $i "oranges”
        ;;
           2)
        echo $i "this code is silly”
        ;;
      esac
done
```