IJNRD.ORG ISSN : 2456-4184
INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD) | IJNRD.ORG
An International Open Access, Peer-reviewed, Refereed Journal

INTRODUCTION TO SHELL SCRIPTING

Author Name: Gunjan Malakar
DESIGNATION: Adhoc Teacher
Address: Amolapatty Chariali, Dibrugarh Assam

Abstract: In its simplest of forms a variable is a placeholder that holds value in computer memory. This value can be a number, text or filename/directory. In the previous unit we have discussed the concept of **shell**; a shell is an interface between the kernel and the user. We can also say that a shell is a command interpreter, which processes commands entered by user on the command line. In UNIX a shell allows a user to create, assign or delete variables. However, variables are temporary which means they are automatically deleted when the shell session is closed. These types of variables which are available only to the current shell are known as shell variables. If we want to make a shell variable persistent, we have to use the concept of environment variable.

Keywords: shell, command interpreter, variable, Kernel, environmental variable.

Concept of Environment Variable

An environment variable is a value that is available widely and can be used by other applications on the UNIX system.

Environment variables can be categorized in three types:

1. Local Environment Variable

Local environment variables are defined only for the current session, they last only for the current session like remote login session or local terminal session. Local environment variables are not specified in configuration files and can be created or removed by special commands.

2. User Environment Variable

As the name suggest user environment variables are defined for a particular user and are executed every time a user logs in using a local terminal session or a remote login session. User environment variables are loaded from the configuration files like .bashrc, .bas_profile, .bash_login, .profile present in the home directory.

3. System wide Environment Variables

The system wide environment variables are available widely for all users in the system. They are basically present in the following directories /etc/environment, /etc/profile, /etc/bash.bashrc, /etc/profile.d. They are loaded every time a user logs in locally or remotely.

The command **env** is used to list all the current environment variables.

Rules for writing Variable Names

Firstly the name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore symbol (_).

By convention, UNIX shell variables will have their names in UPPERCASE.

Example

```
VARIABLE_NAME
VARIABLE_1
_VARIABLENAME
```

***NOTE** we cannot use special characters like **! , *, or –** because they hold a special meaning on the shell.

Variable Declaration

Variables are defined in the following manner –

```
VARIABLENAME=VARIABLEVALUE
```

Example

```
NAME=”GUNJAN MALAKAR”
```

In this example the variable name **NAME** is assigned with the value “GUNJAN MALAKAR”. These types of variables are called **scalar variables;** a **scalar variable** is a variable that can hold only one value at a given point in time.

Accessing Variable Values

We have learnt that a variable is used to store data or information, but in some point of time we need to access the value stored in a variable , so to access the value stored in a variable , we have to prefix the variables name with the dollar sign **($)**

The following example accesses the value of the defined variable **NAME** and prints it on the **STDOUT**

```
NAME= “GUNJAN MALAKAR”

echo $NAME
```

The output of the above example will be −

```
Gunjan Malakar
```

Read-only Variables

UNIX shell allows us to mark a variable read-only, once a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −

```
NAME="Gunjan Malakar"
readonly NAME
NAME="Gunjan"
```

The above example will generate the following result if we try to change the value of variable **NAME** −

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting a variable means directing the shell to remove the variable from the list of variables it is tracking. The unset variable cannot be accessed.

The syntax to unset a defined variable using the **unset** command −

unset variable name

```
NAME="GUNJAN MALAKAR"
unset NAME
echo $NAME
```

The above example does not print anything.

***NOTE** You cannot use the unset command to **unset** variables that are marked **readonly**.

Variable Types

In UNIX there are three main variable types −

Local Variables –

Local variables are present within the current shell instance. They are not available to programs which are started by the shell. Local variables are set at the command prompt.

Environment Variables –

Environment variables, as discussed in the earlier section are available only to the child process of the shell. It can be noticed that environment variables are usually defined by the shell script when they are needed by the programs that it executes.

Shell Variables –

Shell variable as its name suggest is a special variable that is set by the shell for the correct functioning of the shell. Some shell variables are environment variables while others are local variables
The following table shows the special variables in the shell scripts –

SL. NO	VARIABLE	DESCRIPTION
1	**$0**	This variable describes the filename of the current script.
2	**$n**	This variable corresponds to the arguments with which the script was invoked. Here n is a positive number corresponds to the position of an argument
3	**$#**	This variable describes the number of arguments supplied to a script
4	**$***	This variable describes all the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	**$@**	This variable describes all the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	**$?**	This variable describes the exit status of the last command executed.
7	**$$**	This variable describes the process number of the current shell.
8	**$!**	This variable describes the process number of the last background command.

Command-Line Arguments

Command-line arguments also known as command line parameters are arguments that allows the user to control the flow of the command or to specify the input data required for the command. UNIX shell allows users to pass run time argument to commands using command–line arguments. The command-line argument **$0** points to the actual command, program, shell script or function and **$1, $2, $3 ………. $9** are arguments to the command.

Let us consider an example we are considering a file name **file.sh** the contents of the file is “GUNJAN MALAKAR”

```
echo “File Name: $0”
echo “First Parameter: $1”
echo “Second Parameter: $2”
echo “Marked values: $@”
echo “Marked values: $*”
echo “Total number of parameters: $#”
```

The output of the above script is

File Name: ./filename.sh, Here file.sh is the name of the file

```
First Parameter: GUNJAN
Second Parameter: MALAKAR
Marked values: GUNJAN MALAKAR
Marked values: GUNJAN MALAKAR
Total number of parameters: 2
```

Special parameters $* and $@

UNIX shell provides two special parameters **$* and $@** that allows the access of all the command-line arguments once. However, the **$*** special parameter takes the entire list as one argument with spaces between and the **$@** special parameter takes the entire list and separate it into separate arguments.

***Note** Special Parameter **$* and $@** will act same unless they are enclosed in double quotes **“”**.

Exit Status variable $?

After a command is completed, the command returns a numerical value, this value is denoted by **$?** Known as the exit status. Upon successful completion of the command the exit status is set to be numerical 0 and 1 if the command is unsuccessful.

Array

The definition of Array is: an array is a collection of homogenous (similar) elements that are stored in the RAM (random access memory). Values stored in an array are identified using array name with subscripts. Array are single dimensional or two dimensional. An array is used when we are trying to store the name of various students in a class as a set of variables

Let us consider an example of an illustration of array −

```
NAME01="GUNJAN"
NAME02="BHARGAB"
NAME03="SUBHAM"
NAME04="ROKTIM"
NAME05="KABITA"
```

We will use a single array to store the names of all the students mentioned above. To declare an array we will use the following syntax

```
array_name[index]=value
```

Here *array_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands −

```
NAME[0]="GUNJAN"
NAME[1]="BHARGAB"
NAME[2]="SUBHAM"
NAME[3]="ROKTIM"
NAME[4]="KABITA"
```

If you are using the **ksh** shell, the syntax of array initialization is −

```
Set -A array_name value1 value2 ... value n
```

If you are using the **bash** shell, the syntax of array initialization is −

```
array_name = (value1 ... value n)
```

Accessing Array Values

To access any array variable, you can access it as follows -

```
${array_name[index]}
```

Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept −

```
NAME[0]="GUNJAN"
NAME[1]="BHARGAB"
NAME[2]="SUBHAM"
NAME[3]="ROKTIM"
NAME[4]="KABITA"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

The above example will generate the following result −

```
First Index: GUNJAN
Second Index: BHARGAB
```

You can access all the items in an array in one of the following ways −

```
${array_name[*]}
${array_name[@]}
```

Here **array_name** is the name of the array you are interested in. Following example will help you understand the concept −

```
NAME[0]="GUNJAN"
NAME[1]="BHARGAB"
NAME[2]="SUBHAM"
NAME[3]="ROKTIM"
NAME[4]="KABITA"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

The above example will generate the following result −

```
First Method: GUNJAN BHARGAB SUBHAM ROKTIM KABITA
Second Method: GUNJAN BHARGAB SUBHAM ROKTIM KABITA
```

VARIOUS OPERATORS IN UNIX

UNIX supports the following operators −

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

To perform mathematical operations in the shell we will use two external programs **awk** and **expr. expr** is a command line utility in UNIX that evaluates an expression and outputs the corresponding value.

***Note expr** evaluates integer or string expressions, it also includes pattern matching regular expressions.

```
val = ‘expr 2 + 2’
echo “Total value: $val”
```

The following points need to be considered while evaluating the above expression –

- There must be spaces between operators and expression. For example, 2+2 is not correct it should be written as 2 + 2.
- The complete expression should be enclosed between ‘‘(inverted commas).

Arithmetic Operators

The operators that perform arithmetic operations are called Arithmetic Operators.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	Description	Example
+ (Addition)	It is used to perform addition, it adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	It is used to perform subtraction, it subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	It is used to perform multiplication , it multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	It is used to perform division, it divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	It is used to perform modular division it divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give

		0
= (Assignment)	It is used for assignment operator it assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	It is Equality operator it compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	It is a not equality operator it compares two numbers, if both are different then returns true.	[$a != $b] would return true.

***NOTE we should understand that all the conditional expressions should be inside square braces with spaces around them, for example [$a == $b]is correct whereas, [$a==$b] is incorrect. In UNIX shell all the arithmetical calculations are done using long integers.**

Relational Operators

In UNIX Relational operators are used to compare the values of operands to produce a logical value. A logical value is either **true** or **false.** This operator does not work on string values unless the values are numeric.

Assume variable **a** holds 10 and variable **b** holds 20 then –

Show Examples

Operator	Description	Example
-eq	This logical operator checks if the values of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	This logical operator checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	This logical operator checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	This logical operator checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	This logical operator checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	This logical operator checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

***NOTE we should understand that all the conditional expressions should be inside square braces with spaces around them, for example [$a <= $b] is correct whereas, [$a <= $b] is incorrect.**

BOOLEAN OPERATORS

Boolean operators are used to connect or define the relationship between two entities. The three Boolean operators are **AND, OR** and **NOT**. Let us assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
!	The logical negation is used for inverting a true condition to false and false to true.	[! false] is true.
-o	The logical operator OR is true when one of the	[$a -lt 20 -o $b -gt 100] is true.

	operands is true.	
-a	The logical operator AND are true when both the condition are true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

Conditional statements in UNIX shell are used to execute/ transfer the control from one part of the program to another depending on the condition.
There are two types of conditional statements in UNIX.

i. The **if...else** statement

ii. The **case...esac** statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.
Unix Shell supports following forms of **if…else** statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

Test command

The **test** command is a special command in UNIX; it investigates the sort of tests we are performing and translates the result into success or failure. It helps the shell to decide whether to execute the commands in the if block or the commands in the else block.

For example

```
echo Enter a number
read number
if test $number –ge 0
then
echo positive
fi
```

*Note The **test** command is used to carry out Numerical test, String tests and File tests.

File Test Operators

To test the UNIX file properties we use the file test operators. Let us assume a variable **file** that holds the file name "test" and size 100 bytes and it has **read, write** and **execute** permission :-

Examples

Operator	Description	Example
-b file	This command checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	This command checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	This command checks if file is a directory; if yes, then the	[-d $file] is not true.

	condition becomes true.	
-f file	This command checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	This command checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	This command checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	This command checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	This command checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	This command checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	This command checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	This command checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	This command checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	This command checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	This command checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Let us consider an example

```
Enter a file name
read file_name
if [-f $file_name]
then
echo The File exists
else
echo No such file exists
fi
```

In the above example we have use the '[ ]' instead of the test command. This avoids the repeated use of the test keyword, but it is advisable to use a space immediately after '['and immediately before ']'.

String Operators

Array of characters are known as string. In UNIX String operators allow you to manipulate the contents of a variable without resorting to AWK. Some shells such as bash 3.x or ksh93 supports most of the standard string manipulation functions. Standard functions like length, index, and substr are also available. Strings can be concatenated by using double quoted strings. We can ensure that variables exist and set default values for variables and catch errors that result from variables not being set. You can also perform basic pattern matching.

Let us assume variable **a** holds "abc" and variable **b** holds "efg" then –

Operator	Description	Example
=	The equality operator checks if the values of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	The Not Equal operator checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	The –z operator checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	The –n operator checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	The str operator checks if str is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Let us consider an example

```
first_str=”Hello”
second_str=”world”
third_str=
[$first_str=$second_str]
echo $?
[$first_str!=$second_str]
echo $?
[-n $first_str]
echo $?
[-z “$third_str”]
echo $?
[-z $third_str]
echo $?
[“$third_str”]
echo $?
```

In this example we show the various operations that can be tested on strings and also the use of the meta character **$?.** The metacharacter contains the result of the last command whether it is a success (0) or a failure (1). We must observe very carefully that while carrying out the equality test there is a space on either side of “=”.

Nested if-elif Statements

If statements may have another if statement in the true and false block. This type of compound statement is called nested if statement. But, this may not be the best solution when all the branches of if statement depend on a single variable.

The case...esac Statement

To minimize the situation encountered by the if….elif statements UNIX supports the case……..esac statements.

The syntax of **case...esac** statement is given below −

- case...esac statement

Let us consider an example to create a calculator

```
echo "Enter Two numbers: "
read first_number
read second_number

# Input type of operation
echo "Enter Choice :"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read choice

# Switch Case to perform
# calculator operations
case $ch in
  1)result=`echo $first_number + $second_number | bc`
  ;;
  2)result=`echo $first_number - $second_number | bc`
  ;;
  3)result=`echo $first_number \* $second_number | bc`
  ;;
  4)result=`echo "$first_number / $second_number" | bc`
  ;;
esac
echo "Answer : $result"
```

The above example is a simple calculator that calculates the result based on the choice selected by the user for performing the calculation we are using the binary calculator.

Loop Statements

Loop statements are used to execute and repeat a block of statements depending on the value of a condition.

UNIX supports the following loop statements:-

- The while loop

In UNIX shell a **while** loop is executed and repeat a statement block depending on the condition evaluated at the beginning of the loop.

Syntax

While condition

do

statement1

statement2

done

- The for loop

In UNIX shell a **for** loop is used to execute and repeat a statement block depending on a condition which is evaluated at the beginning of the loop.
Syntax
for condition in value
do

```
statement1
statement2
done
```

- The until loop
  In UNIX the **until** loop is same as the **while** loop except that the loop executes until the **TEST-COMMAND** executes successfully.
  Syntax

```
until condition
do
this
and this
done
```

- The select loop
  In UNIX the **select** loop provides a way to create a menu from which users can select options.
  The select loop is useful when you need to ask the user to choose one or more items from a list of choices.

Nesting Loops

Like **if** statements loops can also be nested, which means you can put one loop inside another similar one or different loops. The nesting can go up to unlimited number of times .

Syntax

```
while first command
do
   Statement(s) to be executed if first command is true

   while second command
   do
      Statement(s) to be executed if second command is true
   done

   Statement(s) to be executed if  first command is true
done
```

Example

Here is a simple example of loop nesting.

```
x=0
while [ "$x" -lt 10 ]    # this is loop1
do
   y="$x"
   while [ "$y" -ge 0 ]  # this is loop2
   do
      echo -n "$y "
      y=`expr $y - 1`
   done
   echo
   x=`expr $x + 1`
done
```

The output of the above program is shown below.

***Note echo –n option avoid printing a new line character**

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

The infinite Loop

An infinite loop continues forever if the required condition is not met. This type of loops executes forever without terminating, they executes for an infinite number of times.
Example

In this example the **while** loop is used to display the numbers zero to nine −

```
#!/bin/sh

x=10
until [ $x -lt 10 ]
do
  echo $x
  x=expr $x + 1`
done
```

This loop continues forever because **x** is always **greater than** or **equal to 10** and it is never less than 10.

The break Statement

The **break** statement is used to transfer the control to the end of the statement block or terminate the execution of the entire loop, after the execution of all the lines of code are executed successfully. A **break** is often associated with an **if.**

Syntax

The following **break** statement is used to come out of a loop −

```
break
```

The break command can also be used to exit from a nested loop using this format −

```
break n
```

Here **n** specifies the **n**th enclosing loop to the exit from.

Example

A simple program to display the first 10 natural numbers

```
x=1
while [ $x -lt 10 ]
do
  echo $x
  if [ $x -eq 10 ]
  then
    break
```

```
   fi
   x=`expr $x + 1`
done
```

Upon execution, you will receive the following result −

```
1
2
3
4
5
6
7
8
9
10
```

The continue statement

The **continue** statement is used to transfer the control to the beginning of a statement block in a loop.

Syntax

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

Example

A program to check odd or even −

```
x="1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20"
for y in $x
do
   a=`expr $y % 2`
   if [ $a -eq 0 ]
   then
      echo "even number"
      continue
   fi
   echo "odd number"
done
```

Conclusion

A variable is a placeholder that stores value. A shell is an interface between the kernel and the user. An environment variable is a value that is available widely and can be used by other applications, environment variables are categorized in Local environment variable, user environment variable and system wide environment variable. A local variable is defined only for the current session, user environment variables are used for a particular and system wide are available widely for all users in the system. To access a variable all variables must have a **$** sign as prefix to the variable. In UNIX variables can be of Local variables, environment variables and shell variables. Command-line arguments are arguments that allow user to control the flow of the command or to specify the input data required to the command. UNIX also uses two special parameter **$*** and **$@** that allows the access of all the command-line arguments. An array is a collection of similar elements. To access any array variable we can access it as ${array_name[index]}.

References:

UNIX in a Nutshell by Arnold Robbins (Shroff Publishers and Distributors)
The UNIX Programming Environment by Brian Kernighan &Rob Pike (Prentice-Hall of India)
The Design of the UNIX Operating System by Maurice Bach (Prentice-Hall of India)
UNIX concepts and Applications by Sumitabha Das (The McGraw- Hill Companies)
UNIX Shell Programming Yashavant Kanetkar
Advanced Programming in the UNIX Environment by W. Richard Stevens (Prentice-Hall)
Beginning Linux Programming by Richard Stones and Neil Matthew (Shroff publishers and Distributors)
Advanced UNIX A programmers Guide by Stephen Prata (SAMS)
UNIX Made Easy UNIX and Linux Basics and beyond by John Muster (Tata McGraw Hill)
The UNIX operating system by Kaare Christian, Susan Richter (John Wiley)