Shell::

If user wants any instructions to be given to the system, its done via shell.

* Interface between User and kernel
* Has powerful programming capabilities
* Command Interpreter of Unix

Via shell script we give a list of instruction to the system which interacts with multiple interfaces and the final output is received.

Types of Shell::

* Bourne shell (sh)
* Korn shell (ksh)
* Bourne Again shell (bash)
* POSIX shell (sh)

Vi editor::

$ Vi Filename<enter>

User enters Vi editor

Esc+I or Esc+insert enables user to enter in the Vi editor

Esc:wq 🡺 w is to save and q is to quit

**Vi editor saving  and quitting commands:**

**:w** -Save the contents of the file.

**:q** – Quit from vi editor.

**:q!** -quit from vi editor by discarding any changes.

**:wq** -Save the file and quit from the vi editor.

**Basic navigation**

1. The <h> **key**, in command mode, moves the cursor one character left.
2. The <j> **key**, in command mode, moves the cursor one character down.
3. The <k> **key**, in command mode, moves the cursor one character up.
4. The <l> **key**, in command mode, moves the cursor one character right.

Types of Variables

**#1) Environment variables:**

These are the variables that are visible to the child processes of a shell program.   These include special environment variables that are set by the shell and are required for the shell to function properly.

**Example:**

* $PATH – The set of paths to search for commands.
* $HOME – The path to the current user’s home folder.

**#2) Predefined variables:**

When running commands, the shell expands wildcards, and then assigns the arguments to these predefined variables or ‘positional variables’.

**Example:**

* $0 – The name of the command being executed.
* $1 … $9 – The first to ninth arguments.

set hello how are You

echo $1 , $2

hello how 🡺 Here $1 prints hello and $2 prints how

**#3) User-defined variables:**

These are the variables that are visible to the current instance of the shell.  The ‘export’ command is used to expose local variables to the environment variables.

var1 = 30

echo $var1 🡺 prints 30

Read var2 🡺 asks user to input the value, if user inputs 45

Echo $var2 🡺 prints 45

var3 = 60

echo $var3 🡺 prints 60

readonly $var3 🡺 if readonly is used then the value cant be changed

var3 = 90

echo $var3

Unset command is used to erase a variable

### Operators in Unix

#### **#1) Shell Arithmetic Operators Example**

**These consist of basic mathematical operations:**

* Addition: +
* Subtraction: –
* Multiplication: \*
* Division: /
* Modulus: %

Each of these operators performs the operation on two integer variables or constants.

**For Example, the below program illustrates each of these operations:**

a=20 b =30

$ c=`expr $a + $b`

$ echo “the value of addition=$c”

$ d=`expr $a - $b`

$ echo “the value of subtraction=$d”

$ e= expr $a \\* $b`

$ echo “the value of multiplication=$e”

$ f=`expr $a / $b`

$ echo “the value of division=$f”

$ g= echo `expr $a % $b`

$ echo “the value of modulus=$c”

The Unix shell does not natively support floating point operations(decimals).  A separate command line tool must be used for this.  The ‘bc’ co0mmand is the most standard tool for this.

**Example:**

**A=20.5 b = 50.7**

$ c = `echo “$a + $b” | bc`

$ d = `echo “$a + $b” | bc`

Note that each of the operators needs to be surrounded by a space on both sides, and the ‘\*’ operators need to be escaped with a backslash ‘\’.

#### **#2) Shell Logical Boolean Operators Example**

**The logical operators in Unix are as follows:**

* Not:!
* And: -a
* Or: -o

Unix Conditional Statements The if-elif-fi

Unix provides a number of relational operators in addition to the logical operators mentioned earlier. These can be used to compare numeric values.

* -lt less than
* -le less than or equal to
* -gt greater than
* -ge greater than or equal to
* -eq equal to
* -ne not equal to

**Unix provides a number of ways for conditionally executing the other commands.**

**These are covered below:**

**#1) The if statements**

**Example:**

if <control command>

then

<statements>

fi

**#2) The if…else statements**

**Example:**

if <control command>

then

<statements>

else

<statements>

fi

**#3) The if…elif…else…fi statement**

**Example:**

if <control command>

then

<statements>

elif

then

<statements>

else

<statements

fi

**Given below are some example programs that illustrate these conditional statements:**

**#1) Check if an input number is positive:**

$ echo “Enter a number”

$ read num

$ if [ $num -gt 0 ]

$ then

$ echo “It is a positive number”

$ fi

**#2) Check if an input number is positive or not:**

$ echo “Enter a number”

$ read num

$ if [ $num -gt 0 ]

$ then

$ echo “It is a positive number”

$ else

$ echo “It is not a positive integer”

$ fi

**#3) Check if an input number is positive, zero or negative:**

$ echo “Enter a number”

$ read num

$ if [ $num -gt 0 ]

$ then

$ echo “It is a positive number”

$ elif [ $num -eq 0 ]

$ then

$ echo “num is equal to zero”

$ else

$ echo “It is not a positive integer”

$ Fi

**The Shell Switch Case Syntax and Examples:**

case <word> in

<first pattern>)

<statements>

;;

<second pattern>)

<statements>

;;

\*)

<default statements>

;;

esac

Here, the value of the word expression is matched against each of the choice patterns.  If a match is found then the corresponding statements are executed until the ‘;;’ statement is encountered.  If there is no match, the default statements under ‘\*)’ are executed.

**The following is an Example of a switch case program:**

echo “Enter a number”

read num

case $num in

[0-9])

echo “you have entered a single digit number”

;;

[1-9][1-9])

echo “you have entered a two-digit number”

;;

[1-9][1-9][1-9])

echo “you have entered a three-digit number”

;;

\*)

echo “your entry does not match any of the conditions”

;;

Esac

### Loops in Unix

**You may use different loops based on the situation.**

**They are:**

**#1) Unix For loop statement**

**Example:** This program will add 1+2+3+4+5 and result will be 15

for i in 1 2 3 4 5

do

sum=`expr $sum + $i`

done

echo $sum

**#2) Unix While loop statement**

**Example:** This program will print the value of ‘a’ five times, from 1 to 5.

a=1

while [ $a -le 5 ]

do

echo “value of a=” $a

a=`expr $a + 1`

done

**#3) Unix Until loop statement**

This program will print the value of ‘a’ two times from 1 to 2.

a=1

until [ $a -ge 3 ]

do

echo “value of a=” $a

a=`expr $a + 1`

done

While running these loops, there may be a need to break out of the loop in some condition before completing all the iterations or to restart the loop before completing the remaining statements. This can be achieved with the ‘break’ and ‘continue’ statements.

**The following program illustrates the ‘break’ operation:**

num=1

while [ $num -le 5 ]

do

read var

if [ $var -lt 0 ]

then

break

fi

num=`expr $num + 1`

done

echo “The loop breaks for negative numbers”

* **grep:** Find lines in stdin that match a pattern and print them to stdout.
* **sort:** Sort the lines in stdin, and print the result to stdout.
* **uniq:** Read from stdin and print unique (that are different from the adjacent line) to stdout.
* **cat:** Read lines from stdin (and more files), and concatenate them to stdout.
* **more:** Read lines from stdin, and provide a paginated view to stdout.
* **cut:** Cut specified byte, character or field from each line of stdin and print to stdout.
* **paste:** Read lines from stdin (and more files), and paste them together line-by-line to stdout.
* **head:** Read the first few lines from stdin (and more files) and print them to stdout.
* **tail:** Read the last few lines from stdin (and more files) and print them to stdout.
* **wc:** Read from stdin, and print the number of newlines, words, and bytes to stdout.
* **tr:** Translate or delete characters read from stdin and print to stdout.