Linux Shell Scripting

CSC 1153 – LABORATORY ASSIGNMENTS

K.D.C.G Kapugama
Department of Computer Science
Faculty of Science
University of Ruhuna

What is a Shell Script?

- A text file containing commands that are executed in a sequential way by the shell interpreter.
- The goal of shell scripting is to automate the execution a series of tasks.
- The first line of a shell script must specify the shell that is used to execute the commands.

```
#!/bin/bash
```

Creating a Shell Script

- A text editor should be used (e.g. nano, gedit, vim).
- Once the script is created, it should be made executable
  - `chmod u+x <<script_name>>`
- Launching the script:
  - `./<<script_name>>`

Creating a Shell Script

- **Example:**

```
#!/bin/bash
echo “Hello World”
pwd
echo “$SHELL”
```

Displaying Messages

- **“echo”** command is used.
  - E.g.
    - `echo Hello World !`
    - `echo “Let’s see who logged into the system”`
  -

Variables

- Assigning a value to a variable
  - `<<variable_name>>=<<value>>`
  - Example:
    - `myvar=”first”`
    - `mynum=200`
- Using the value of a variable:
  - Use ‘$’ sign at the beginning of the variable name
    - `echo $myvar`
    - `echo “The assigned number is $mynum”`

Variables

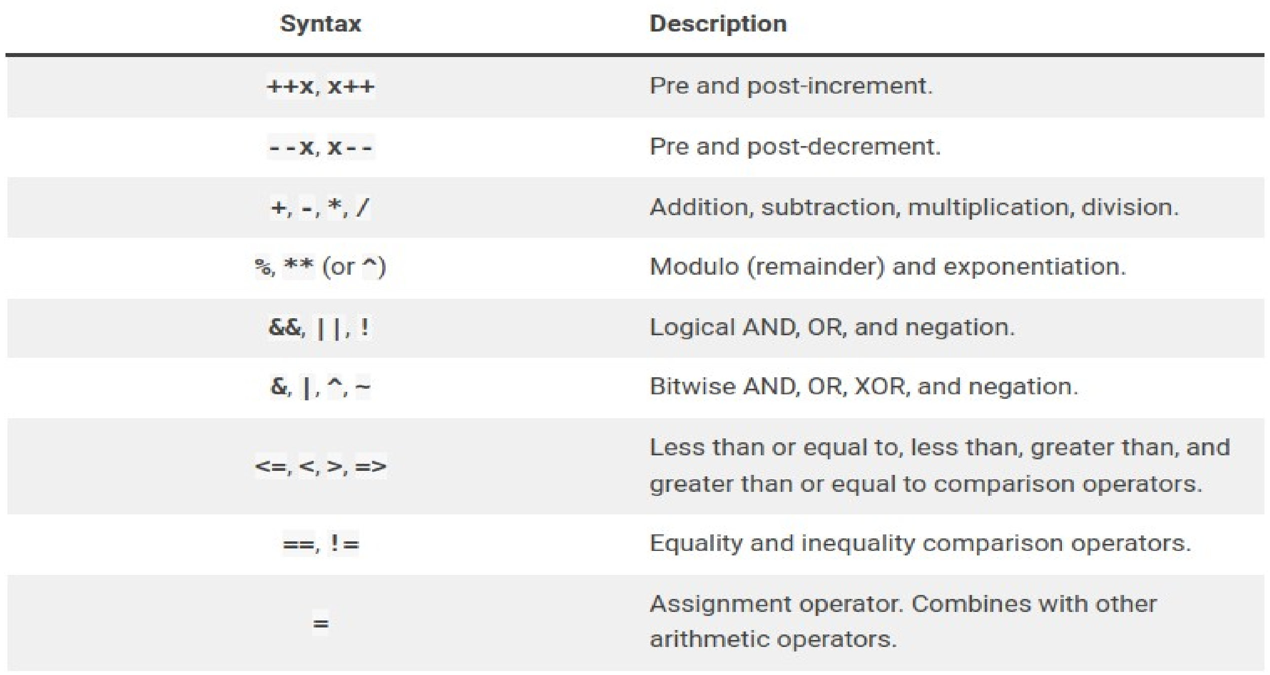
- ` (backtick)： Assign the output of a shell command to a variable.
  - E.g. `cur_date=`date``

Arithmetic Expressions

- Double Parenthesis:
  - E.g.: `total=$(($var1+$var2))`
- **expr** command with backtick
  - Eg : ``total=`expr $var1 + $var2` ``
  - * should be used for multiplication

- Floating point arithmetic
  - Use bc command
  - E.g. ``total = `echo “$var1 * $var2” | bc` ` ``

Arithmetic Expressions



Syntax	Description
++x, x++	Pre and post-increment.
--x, x--	Pre and post-decrement.
+, -, *, /	Addition, subtraction, multiplication, division.
%, ** (or ^)	Modulo (remainder) and exponentiation.
&&, ||, !	Logical AND, OR, and negation.
&, |, ^, ~	Bitwise AND, OR, XOR, and negation.
<=, <, >, =>	Less than or equal to, less than, greater than, and greater than or equal to comparison operators.
==, !=	Equality and inequality comparison operators.
=	Assignment operator. Combines with other arithmetic operators.

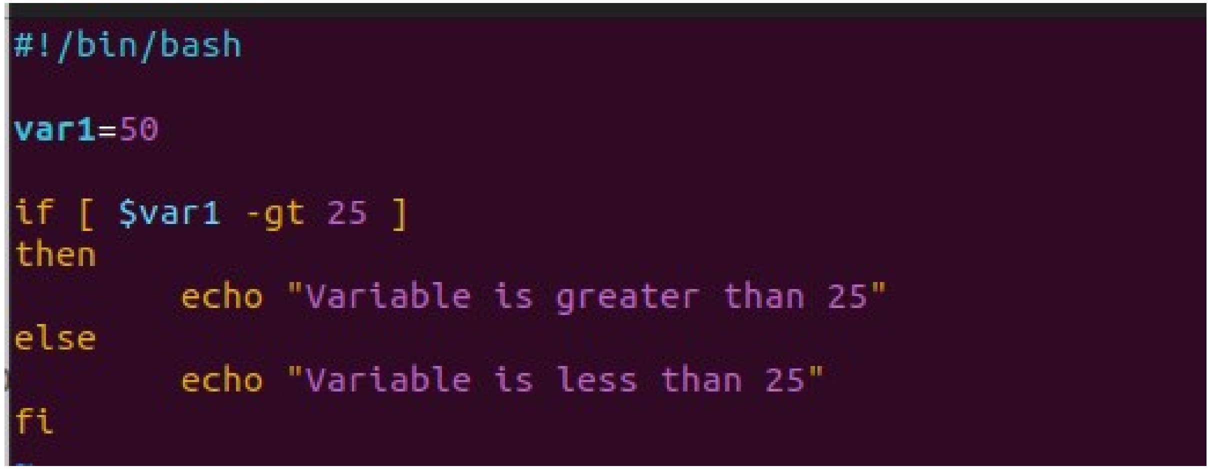
If command

- **Syntax**

```
if [ <condition> ]
then
      <commands if condition is true>
else
      <commands if condition is false>
fi
```

If command

- **Example:**



```
#!/bin/bash

var1=50

if [ $var1 -gt 25 ]
then
        echo "Variable is greater than 25"
else
        echo "Variable is less than 25"
fi
```

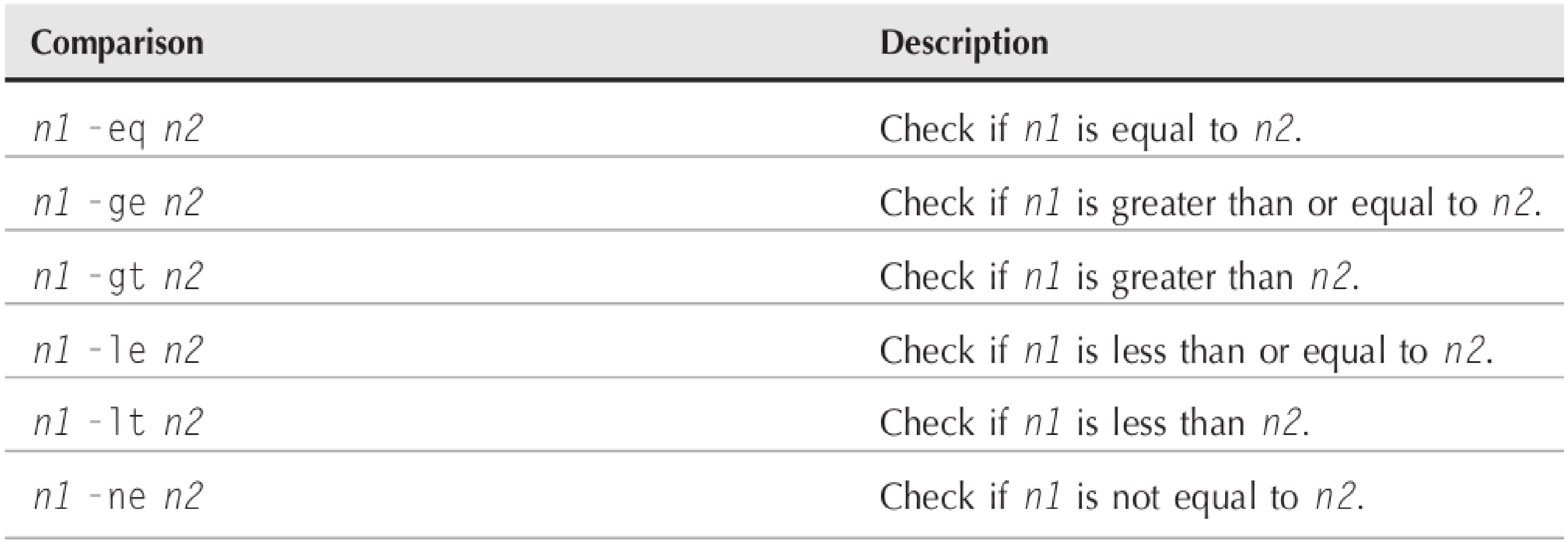
- The script will print “variable is greater than 25”

Nested ifs

- **Syntax**

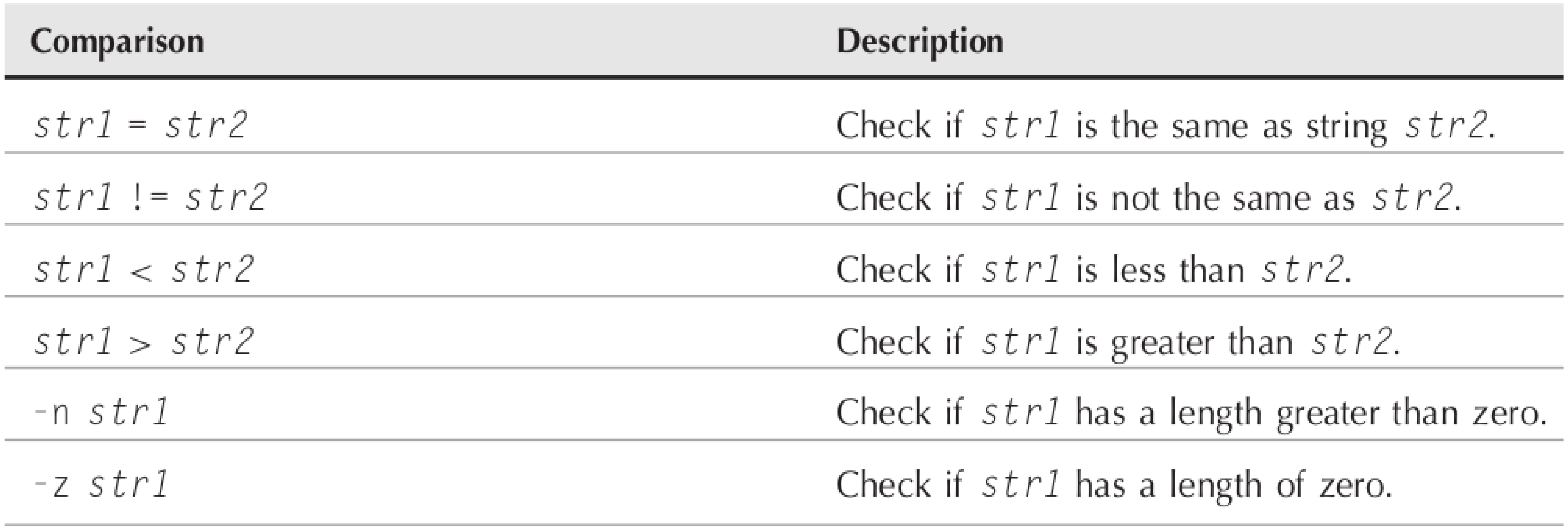
```
if [ <condition> ]
then
     <commands if condition is true>
elif [ <condition> ]
then
     <commands elif condition is true>
else
     <commands if condition is false>
fi
```

Numeric Comparisons



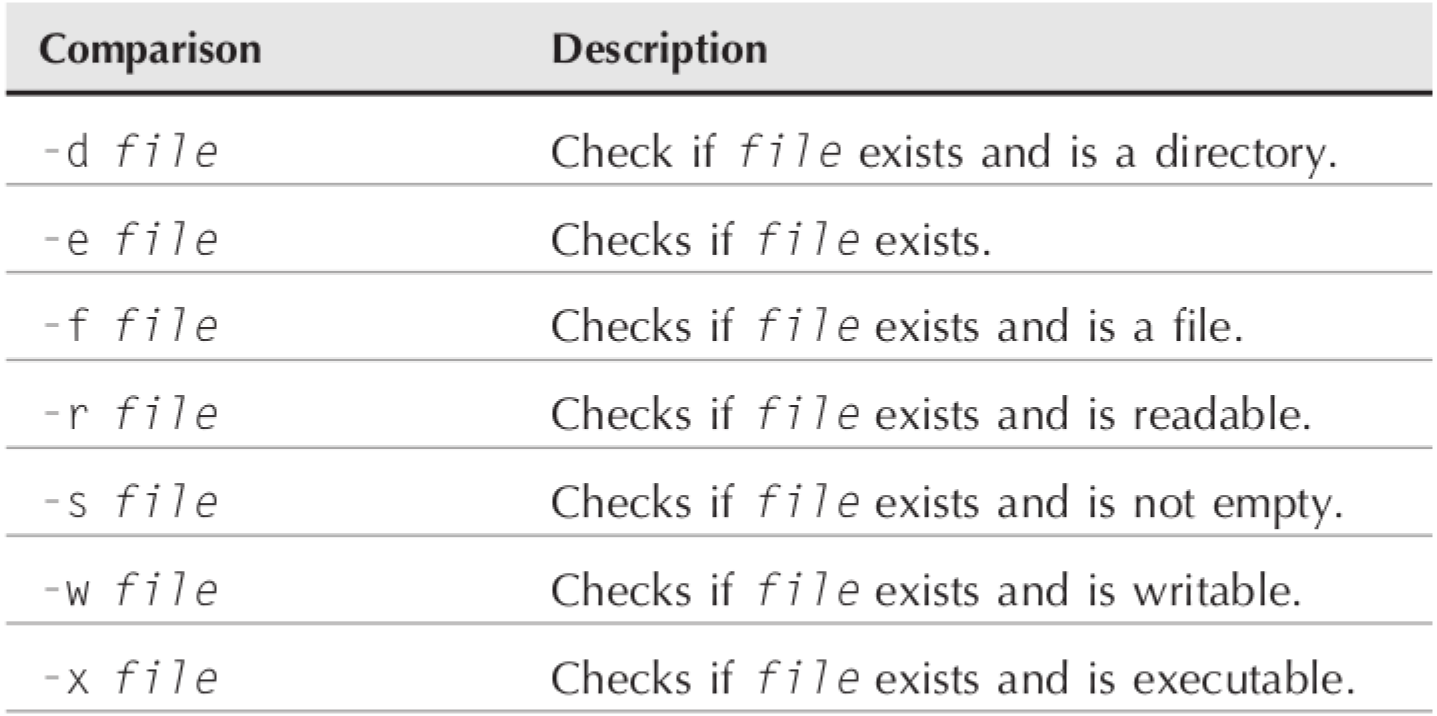
Comparison	Description
n1 -eq *n2*	Check if *n1* is equal to *n2*.
n1 -ge *n2*	Check if *n1* is greater than or equal to *n2*.
n1 -gt *n2*	Check if *n1* is greater than *n2*.
n1 -le *n2*	Check if *n1* is less than or equal to *n2*.
n1 -lt *n2*	Check if *n1* is less than *n2*.
n1 -ne *n2*	Check if *n1* is not equal to *n2*.

String Comparisons



Comparison	Description
`str1 = str2`	Check if `str1` is the same as string `str2`.
`str1 != str2`	Check if `str1` is not the same as `str2`.
`str1 < str2`	Check if `str1` is less than `str2`.
`str1 > str2`	Check if `str1` is greater than `str2`.
`-n str1`	Check if `str1` has a length greater than zero.
`-z str1`	Check if `str1` has a length of zero.

File Comparisons



Comparison	Description
-d *file*	Check if *file* exists and is a directory.
-e *file*	Checks if *file* exists.
-f *file*	Checks if *file* exists and is a file.
-r *file*	Checks if *file* exists and is readable.
-s *file*	Checks if *file* exists and is not empty.
-w *file*	Checks if *file* exists and is writable.
-x *file*	Checks if *file* exists and is executable.

Complex Conditions

- Logical operators:
  - || : Logical OR
  - && : Logical AND
- Syntax:
  - [<condition 1>] || [<condition 2>]
  - [<condition 1>] && [<condition 2>]

Case Command

- Syntax

```
case <<variable>> in
pattern 1)  command1;;
pattern 2) command 2;;
pattern 3 | pattern 4) command 3;;
*) default commands
esac
```

While Loop

- While a given condition is true, a block of commands is repeated.
- **Syntax**

```
while [<condition>]
do
      <block of commands>
done
```

For Loop

- **Syntax**

```
for <variable> in <list>
do
        <block of commands>
done
```

- **c-style**

```
for (( i=1; i <= 10; i++ ))
do
        echo "The next number is $i"
done
```

Declaring Arrays

- `<<array_name>>= ( <<elements separated by spaces>>)`
  - E.g. : `my_ar=( “cat” “dog” “mouse” “elephant”)`
  - Referring to a particular element : `${<array_name>[<index>]}`
  - Getting the all elements of an array: `${<array_name>[@]}`
  - Getting the array size: `${#<array_name>[@]}`
-

Declaring Arrays

- Looping through an array:

```
for str in ${my_ar[@]}
do
  echo "$str"
done
```

- Loop with indexes

```
for i in ${!my_ar[@]}
do
  echo “element $i is ${my_ar[$i]}”
done
```

Command Line Arguments

- Example:
  - ➢ `./script.sh <args>`
- `$0` : name of the script (script.sh)
- `$1, $2, $3 …` : arguments
- `$@` : All arguments
- `$#` : number of arguments

Creating Functions

```
function <name>
{
        commands
}
```