**Georgia Southern University**

**Allen E. Paulson College of Engineering and Computing**

**Information Technology Department**

**Dr. Atef Shalan**

**Course: IT2530-Operating Systems**

**Module 7-** Shell Scripting– reading material

Contents and Reading Material

[**7** Shell Scripting 1](#_Toc91077152)

[7.1 Types of Shell 2](#_Toc91077153)

[7.2 Shell Commands and Utilities 3](#_Toc91077154)

[7.3 Basic shell commands 3](#_Toc91077155)

[7.3.1 Text File Manipulation 4](#_Toc91077156)

[7.3.2 File system Management 4](#_Toc91077157)

[7.3.3 Process Management 4](#_Toc91077158)

[7.3.4 Network Management 5](#_Toc91077159)

[7.3.5 System Management 5](#_Toc91077160)

[7.3.6 User Management 6](#_Toc91077161)

[7.4 Shell Script Structure 6](#_Toc91077162)

[7.5 Variables, Operators, and Logic Structures 6](#_Toc91077163)

[7.6 Shell Logic Structures 9](#_Toc91077164)

[7.6.1 Conditional statements 10](#_Toc91077165)

[7.6.2 Loop statements 11](#_Toc91077166)

[7.7 Shell Functions 12](#_Toc91077167)

# Shell Scripting

A shell script is a sequence of commands that users have a repeated use for it. The Unix/Linux “Shell” is a system program that provides access to some OS kernel functions through the command-line interface. User commands can be entered into the shell program in two ways: directly through the shell window or from a file called the shell script file. Shell program interprets script commands. I.e., it reads commands from the script line by line and executes them.

## Types of Shell

A Shell Script program is analogs to a programming language. The following are the most common shell types:

* **sh**:
  + Name: **Bourne Shell**
  + The original shell
  + By Steven Bourne of AT&T
  + Not the standard shell
  + Still used on some UNIX/Linux systems.
  + This is the basic shell, a small program with few features.
  + Use prompt of $
* **bash**:
  + Name: **Bourne Again shell**
  + GNU Improved Bourne Shell
  + The default for all Unix versions
  + A superset of the Bourne shell with various add-ons and plug-ins
  + the standard shell for common users
  + Intuitive, flexible, and easy to learn.
  + A powerful tool for advanced scripting and professional user.
  + Use prompt of $
* **csh**:
  + Name: C Shell
  + C-like script syntax
  + *By Bill Joy of Univ. of California*
  + Used by C language programmers.
  + Use prompt of %
* **ksh**:
  + Name: Korn shell
  + A superset of the Bourne Shell and C Shell;
  + By David Korn of AT&T
  + Very hard for beginners.
  + Use prompt of $

To learn about the supported shells on a Linux system, you execute the following command.

$ cat /etc/shells

To know what shell you currently use

$ echo $SHELL

To switch shell: type the shell name, examples:

$ bash

$ exec bash

## Shell Commands and Utilities

When the shell program executes a command, bash creates a new bash process using a [fork method](https://www.section.io/engineering-education/fork-in-c-programming-language/). Some shell commands do not create new processes: built-in commands. [Built-in commands](https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf) are not Kernel commands but rather, they are implemented as part of the shell program itself. Unlike kernel commands, a built-in command is executed directly, without creating a new process. Examples of Bash built-in-commands are:

:, ., break, cd, continue, eval, exec, exit, export, hash, pwd, return, set, shift, test, times, trap, and unset, alias, bind, command, declare, echo, enable, help, let, local, logout, read, type, typeset, and unalias

The [mysystem.sh file](https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf) (by Machtelt Garrels, Bash-Beginners-Guide) is a shell example that executes some commands (date, w, uname, uptime) to display information about you and your machine.

1 #!/bin/bash

2 clear

3 echo "This is information provided by mysystem.sh. Program starts now."

4 echo "Hello, $USER"

5 echo

6 echo "Today's date is `date`, this is week `date +"%V"`."

7 echo

8 echo "These users are currently connected:"

9 w | cut -d " " -f 1 - | grep -v USER | sort -u

10 echo

11 echo "This is `uname -s` running on a `uname -m` processor."

12 echo

13 echo "This is the uptime information:"

14 uptime

## Basic shell commands

Unix commands allow to navigate and control Unix/Linux system and manage user files or directories. The main categories of Unix/Linux commands are the following.

* Text File Manipulation
* Filesystem Management
* Process Management
* Network Management
* System Management
* User Management

The following paragraphs include the main commands in each category

### Text File Manipulation

Text file manipulation commands help inspect, view, and process text files (e.g., configuration files and data files).

cat –display file contents on the standard output

more/less: view text file on separate pages

head/tail: show the first (or last) a specified number of lines of a text file

grep: search for text patterns in text files

cmp: compare two files.

cut: removes part of a file.

File viewing and exiting

vi: command-based text editor

gedit/emacs: GUI text editors

### File system Management

File system management commands allow managing files and access permissions, they also allow navigating filesystems and managing directories.

File operation management commands

ls: list files and directories of the current folder

cp: copy files between folders

rm: remove files and/or directories

mv: move files from a folder to another, also used for renaming files

chmod: change file access permissions

chown: change file ownership

Directory management commands

cd: change the current directory

pwd: display current directory location

ln: make links to files and directories

mkdir: make a new directory

rmdir: remove a directory

### Process Management

Process management commands allow listing executing processes and viewing and controlling their status.

man: display manual or help pages for a specified command

ps: lists current processes

top: shows a real-time view of the running system processes

kill: kill a process

history: show previously used shell commands

### Network Management

Network management commands allow inspecting network setup and exploring network connections. These commands also allow remote access to the system and system resources.

Networking commands in Unix

Ifconfig/ip: show and configure IP address information

ping: check whether a remote host is reachable via ICMP ping or not

netstat: displays network connections and routing information

Remote access commands

telnet: remote access protocol and utility based on clear text (insecure, no encryption).

ssh: Secure remote access utility

File transfers commands

ftp –File Transfer Protocol and client utility based on clear text (insecure)

sftp: secure file transfer protocol and client utility

wget: retrieves content from remote servers via HTTP/HTTPS and FTP protocols

### System Management

System management commands allow configuring hostname and viewing/setting system information:

Unix system status commands

hostname: show or set a computer hostname

uname: prints basic system information

w: show who is logged on the system and lists their activities

Time and Date commands

date: show current date and time of the computer

sleep: wait for a specified time (in seconds)

uptime: show how long the system has been up

Shutdown and Reboot commands

shutdown –shut the system down gracefully

halt: ungraceful shutdown kills system OS services and does not stop it gracefully.

reboot: an ungraceful reboot of the system without stopping its services

### User Management

User management commands allow viewing and configuring user information.

whoami: show the currently active user

groups: show the groups of the current user

passwd: change the user password

last: show the history of user logins into the system

## Shell Script Structure

In this section, we learn about the shell script program structure. A shell is typically executed by entering the name of the script file on the command line. To create a shell script, you may open a new file in a text editor such as vi, emacs, gedit. The script file name cannot conflict with existing commands and is recommended to end in .sh (*e.g., ~/myScripts*). To be able to execute the script, it should have *execute* permission. To do that, you may run the following command:

chmod u+x script1.sh

After saving your script into specific folder (*e.g., ~/scripts*), you can execute it as follows

./myScript.sh

To get a shell script accessible from any storage location, you may add the script file path (*e.g., ~/scripts*) into the PATH environment variable as follows

export PATH="$PATH:~/scripts"

Comments are parts of a shell script which does not affect its execution. Comments start with a # character. Comments are mainly used to identify script information to make code easy to read by other users and to record timely modifications.

## Variables, Operators, and Logic Structures

A shell script program can include commend lines, logic structure, variables, and operators. Shell variables are symbolic names that help keep values in memory for later use. Operators can be used in scripts to allow performing mathematical, logical, string operations, and other operations.

There are three types of variables

* **Global Variables**:
  + Global variables (AKA environment variables) are available in all shells.
  + They can be displayed using env or printenv commands
  + Examples are

CC=gcc

CDPATH=.:~:/usr/local:/usr:/

SHELL=/bin/bash

* **Local Variables**
  + Local variables are only available in the current shell.
  + They can be set and displayed using the **set** command.
  + Examples are

COLUMNS=65

GROUPS=()

HERE=/home/adam

A variable is created when their values are set in an assignment statement, examples:

VARNAME="value"

welcome=’Hello’

Variable contents are accessed using ‘$’, Example viewing the value of a variable is:

echo $welcome # will print Hello

The above variables are local and only available in the current shell, i.e., child processes of the current shell will not see these variables. Exporting the variables allows passing these variables to the subshells. Variables that are exported are referred to as environment variables. Setting and exporting variables are usually done in one step as follows:

export VARNAME="value"

export welcome=’Hello’

Some parameters have specific meaning in shell scripting. These parameters may only be referenced but not assigned. Examples of these parameters are:

* $0: a parameter that contains the script filename
* $1, $2, …, $n: a sequence of positional command line parameters, 1st, 2nd 3rd respectively.
* $#: number of command-line parameters
* $$: process ID of the shell window program
* $\*: expansion of the positional parameters, starting from one.
* $@: expansion of the positional parameters as a loop of one parameter at a time
* $?: exit the code of the last command

The original code of the following example can be found in [Machtelt Garrels book](https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf).

#!/bin/bash

# positional.sh

# This script reads 3 positional parameters and prints them out.

POSPAR1="$1"

POSPAR2="$2"

POSPAR3="$3"

echo "$1 is the first positional parameter, \$1."

echo "$2 is the second positional parameter, \$2."

echo "$3 is the third positional parameter, \$3."

echo

echo "The total number of positional parameters is $#."

By executing this code we get the following

user ~> positional.sh one two three four five

one is the first positional parameter, $1.

two is the second positional parameter, $2.

three is the third positional parameter, $3.

The total number of positional parameters is 5.

user ~> positional.sh one two

one is the first positional parameter, $1.

Some words have special meaning to the shell program as reserved words. To remove the special meaning of an assigned parameter value, quoting is used. [Quoting](https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf) uses specific characters to introduce a different meaning to the string value as follows.

* **Escape character**: uses a non-quoted backslash character (\) to remove the special meaning from a single character, e.g.,

myVar = 1

echo $myVar # will print 1

echo \$myVar # will print $myVar. This is because the $ lost its shell scripting meaning.

* **Single quotes**: preserve the literal value of each character enclosed within the two quotes, e.g.,

echo '$data' # will print $data, not the value in it.

* **Double quotes**: preserve the literal value of all character enclosed except for $, `, and \, e.g.,

$age = 23

echo “Adam is $age years old” # will print Adam is 23 years old.

Shell Script can capture user input during execution. To read from the standard input device we use the read command, e.g.,

echo “Who are you?”

read username

echo “Hello $username”

[Shell expansion](https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf) is a mechanism that interprets or resolves the command tokens into other meaningful tokens based on the type of expansion. There are mainly eight kinds of expansion in shell scripting. Understanding these expansion help script programmers to design and interpret script commands and their parameters.

* Brace expansion: allows generating arbitrary strings using a pattern.

echo person\_{1,2,3}\_age # will print person\_1\_age person\_2\_age person\_3\_age

* Tilde expansion: specifies a tilde-prefix (enclosed between the first character ~ and the following /), which has a meaning based on the tilde-prefix type.

~ # will be interpreted to the value of $HOME

~/foo # will be interpreted to $HOME/foo

~fred/foo # will be interpreted to the subdirectory foo of the home directory of the user Fred

~+/foo # will be interpreted to $PWD/foo

~-/foo # will be interpreted to ${OLDPWD-'~-'}/foo

* Shell parameter and variable expansion: The "$" character introduces parameter expansion, command substitution, or arithmetic expansion by substituting the value of parameter or variable.

echo $SHELL # will print / bin/bash

* Command substitution: run a shell command and store its output to a variable or display back using echo command.

echo "Today is $(date)" # will print today's date, note date is a command

echo "Today is `date`" # will do the same

* Arithmetic expansion: allows evaluating an arithmetic expression substituting its result. E.g.,

Var++ # variable post-increment

Var1 + var2 # addition

Var1 \* var2 # multiply

Var1 \*\* var2 # exponentiation

Var1 > var2 # greater

Var1 <= var2 # less or equal

Condition1 && Condition2 #logical AND

Condition1 !! Condition2 # logical OR

expr ? expr : expr # conditional evaluation

# etc.

* Process substitution: passes the output of one process to the input of another (pipelining)

who | wc −l # the output of who will be passed to wc -l command to count the number of lines.

* Word splitting: treats each of the [Internal Field Separators in $IFS](https://linuxhint.com/use-ifs-in-bash/) as a delimiter and splits the results into separate words.

Str = “this is any string with some words”

IFS = ‘ ‘

read -a words <<< “$str”

echo “there are ${#strarr[\*]} words in the given string”

# will print there are 7 words in the given string

* File name expansion: allow using special characters as wildcards for matching filenames

\*.txt # all files with .txt extension

A??.txt # all files with three letters that starts with a and that have .txt extension

Details of these shell expansions are required in this module and their details can be found in section 3.4 of [Machtelt Garrels book](https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf).

## Shell Logic Structures

Logic structures provide support sequential flow (sequential logic flow) of commands, branching from one point in a script to another (Conditional branching), and repeating a script part for several times (loop). The following components are the main parts of a shell script structure:

* **Sequential Logic:** execute commands in the order in which they appear in the shell program.
* **Conditional branch:** a code point in the program when a logical choice determines the next logical flow of the program.
* **Case**: conditional statement with multiple branches
* **Loop**: a part of the shell program code that is executed zero or more times.

### Conditional statements

Conditional (if) statements allow different courses of action to be taken depending on, depending on the success or failure of a command or a test condition. If statement can have the following syntax:

**if** command1 # executes successfully

**then**

command\_a # will be executed

**elif** command2 #executes successfully

**then**

command\_b # will be executed

command\_c # will be executed as well

**else**

command\_d # default command to be executed

**fi**

# elif and else clauses are optional, and they can be omitted.

# many elif parts can exist.

Example code

#!/bin/bash

# Basic if statement

if [ $1 -gt 1000 ]

then

echo You enetered a large number.

fi

Example 2

if date | grep “Mon”

then

echo “Let\’s go to work!”

fi

Test conditions can include logical operators. The following are some example text conditions with logical operators.

str1 == str2 # True if the two strings are identical

str1!= str2 # True if strs are different

num1 –eq num2 # True if numbers are equal

num1 –ne num2 # not equal

num1 –lt num2 # Less than

num1 –gt num2 #Greater than

num1 –le num2 #Less than or equal

num1 –ge num2 #Greater than or equal

Example:

if `date +’%H’` -gt 17

then

echo “Time to go home!!!”

fi

Logical conjunction can combine Boolean expressions as follows:

if date | grep “Fri” && `date +’%H’` -gt 17

then

echo “Welcome to the weekend!!!”

fi

Primaries are expressions that make up a test command. The following are some of the common primaries.

-a file Test if file a file exists

-d file Test if file is a directory

-f file Test if file is a regular file and not a directory

-s file Test if the file is not empty

-r file Test if the file is readable

-w file Test if the file is writable

-x file Test if the file is executable

Example:

F=/filepath/filename

if [[ -a $F ]]; then

print "$F exists"

else

print "$F does not exist "

fi

### Loop statements

A loop statement allows a block of code to be repeated a number of times. Loop iterations can be pre-determined (for loop) or conditional (while loop).

The syntax of the for statement is as follows

for var in list do

command\_1

command\_2

# ...

Command\_n

done

Example:

for num in 0 1 2 3 4 do

echo "$num X 10 = $(( $num \* 10 )) "

done

Example 2:

ls > list

for i in `cat list` do

cp "$i" "$i".bak

done

The syntax of the while loop is as follows

while command # execute successfully

do

command\_1

command\_2

# ...

Command\_n

done

Example:

i="100"

while [ $i -gt 0 ]

do

$i--

echo "counting backward $i"

done

## Shell Functions

Shell functions provide a method to identify a part of the code and reuse it when needed. The name of the function must be unique within the shell or script.

The syntax of the function is as follows:

Function\_name

{

command\_1;

command\_2;

# ...

Command\_n;

}

Example:

sayHello () {

echo "Hello there"

}

Hello # Call or invoke your function

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_