Amelia Rotondo
CWID: 887925113
CPSC 254-01
9/13/2023

CPSC 254 Lab #4
Shell Scripting
(Originally prepared by David Heckathorn, modified by Shivansh Vijay Nathani, Tejaas Reddy)

Much administration in Linux is made easier through scripts.

Recall that a shell script begins with bang "!"
Bash supports integer arithmetic with the let builtin.

```
$ let x=1+1
$ echo $x
```

test

Bash scripts frequently use the [(a synonym for test) shell builtin for the conditional evaluation of expressions. test evaluates an expression and exits with either status code 0 (true) or status code 1 (false).

test supports the usual string and numeric operators, as well as a number of additional binary and unary operators which don't have direct analogs in most other programming languages. You can see a list of these operators, along with other useful information, by entering help test in your shell. The output of this is shown below. Note that help is similar to man, except it is used for bash functions instead of other programs.

```
$ help test
test: test [expr]
  Exits with a status of 0 (true) or 1 (false) depending on
  the evaluation of EXPR.  Expressions may be unary or binary.  Unary
  expressions are often used to examine the status of a file.  There
  are string operators as well, and numeric comparison operators.

  File operators:

    -a FILE       True if file exists.
    -b FILE       True if file is block special.
    -c FILE       True if file is character special.
    -d FILE       True if file is a directory.
    -e FILE       True if file exists.
    -f FILE       True if file exists and is a regular file.
    -g FILE       True if file is set-group-id.
    -h FILE       True if file is a symbolic link.
    -L FILE       True if file is a symbolic link.
    -k FILE       True if file has its `sticky' bit set.
    -p FILE       True if file is a named pipe.
    -r FILE       True if file is readable by you.
    -s FILE       True if file exists and is not empty.
    -S FILE       True if file is a socket.
    -t FD         True if FD is opened on a terminal.
    -u FILE       True if the file is set-user-id.
```

```
      -w FILE       True if the file is writable by you.
    -x FILE       True if the file is executable by you.
    -O FILE       True if the file is effectively owned by you.
    -G FILE       True if the file is effectively owned by your group.
    -N FILE       True if the file has been modified since it was last
                                    read.

  FILE1 -nt FILE2  True if file1 is newer than file2 (according to
                modification date).

  FILE1 -ot FILE2  True if file1 is older than file2.

  FILE1 -ef FILE2  True if file1 is a hard link to file2.

 String operators:

    -z STRING     True if string is empty.

    -n STRING
    STRING        True if string is not empty.

    STRING1 = STRING2
                True if the strings are equal.
    STRING1 != STRING2
                True if the strings are not equal.
    STRING1 < STRING2
                True if STRING1 sorts before STRING2 lexicographically.
    STRING1 > STRING2
                True if STRING1 sorts after STRING2 lexicographically.

 Other operators:

    -o OPTION     True if the shell option OPTION is enabled.
    ! EXPR        True if expr is false.
    EXPR1 -a EXPR2 True if both expr1 AND expr2 are true.
    EXPR1 -o EXPR2 True if either expr1 OR expr2 is true.

    arg1 OP arg2   Arithmetic tests.  OP is one of -eq, -ne,
                -lt, -le, -gt, or -ge.

 Arithmetic binary operators return true if ARG1 is equal, not-equal,
 less-than, less-than-or-equal, greater-than, or greater-than-or-equal
 than ARG2.
```

We can test integer equality

```
$ [ 0 -eq 0 ]; echo $? # exit code 0 means true
0
$ [ 0 -eq 1 ]; echo $? # exit code 1 means false
1
```

string equality

```
$ [ zero = zero ]; echo $? # exit code 0 means true
0
$ [ zero = one ]; echo $? # exit code 1 means false
1
```

and a number of other string and numeric operations which you are free to explore.

Flow Control

bash includes control structures typical of most programming languages – if-then-elif-else, while for-in, etc. You can read more about conditional statements and iteration in the Bash Guide for Beginners from the Linux Documentation Project (LDP). You are encouraged to read those sections, as this guide provides only a brief summary of some important features.

if-then-elif-else

The general form of an if-statement in bash is

```
if TEST-COMMANDS; then

  CONSEQUENT-COMMANDS

elif MORE-TEST-COMMANDS; then

  MORE-CONSEQUENT-COMMANDS

else

  ALTERNATE-CONSEQUENT-COMMANDS;

fi
```

Indentation is good practice, but not required.

For example, if we write

```
#!/bin/bash
# contents of awesome_shell_script

if [ $1 -eq $2 ]; then
  echo args are equal
else
  echo args are not equal
fi
```

we see

```
$ ./awesome_shell_script 0 0
args are equal
$ ./awesome_shell_script 0 1
args are not equal
```

while

The general form of a while loop in bash is

```
while TEST-COMMANDS; do

 CONSEQUENT-COMMANDS

done
```

If TEST-COMMANDS exits with status code 0, CONSEQUENT-COMMANDS will execute. These steps will repeat until TEST-COMMANDS exits with some nonzero status.

For example, if we write

```
#!/bin/bash
# contents of awesome_shell_script

n=$1
while [ $n -gt 0 ]; do
 echo $n
 let n=$n-1
done
```

we see

```
$ ./awesome_shell_script 5
5
4
3
2
1
```

Functions

bash supports functions, albeit in a crippled form relative to many other languages. Some notable differences include:

- Functions dont return anything, they just produce output streams (e.g. echo to stdout)
- bash is strictly call-by-value. That is, only atomic values (strings) can be passed into functions.
- Variables are not lexically scoped. bash uses a very simple system of local scope which is close to dynamic scope.
- bash does not have first-class functions (i.e. no passing functions to other functions), anonymous functions, or closures.

Functions in bash are defined by

```
name_of_function() {

 FUNCTION_BODY

}
```

And called by

```
name_of_function $arg1 $arg2 ... $argN
```

Note the lack of parameters in the function signature. Parameters in bash functions are treated similarly to global positional parameters, with $1 containing the $arg1, $2 containing $arg2, etc. For example, if we write

```
#!/bin/bash
# contents of awesome_shell_script

foo() {
  echo hello $1
}

foo $1
```

we see

```
$ ./awesome_shell_script world
hello world
```

Assignment

Write a shell script phonebook which has the following behavior:

•./phonebook new name number: adds an entry to the phonebook. and

•./phonebook name: displays the name and phone number associated with that name.

•./phonebook list: displays every entry in the phonebook (in no particular order). If the phonebook has no entries, display phonebook is empty

•./phonebook remove name: deletes the entry associated with that name.

•./phonebook clear: deletes the entire phonebook. - https://decal.ocf.berkeley.edu/labs/b3

Hint* You can find a lot of shell script examples at https://linuxhint.com/30_bash_script_examples/