CE350
Lecture6

PROGRAMMING THE BASH SHELL
PART II

by İlker Korkmaz and Kaya Oğuz

Programming the bash

- As pointed in syllabus document, there will be 5 lecture weeks to cover the programming with bash scripts.
- This lecture is the second part on scripting in bash.

References

- The contents of this lecture are prepared with the help of and based on:
  - the textbook, UNIX Shells by Example (Chapter14)
  - the tutorial, Advanced Bash-Scripting Guide at http://tldp.org/LDP/abs/html/

Contents of Lecture6

Among the Bash programming concepts:

- Reading User Input
- Command Line Arguments
- Conditionals
- Loops
- Functions

Some reminders:

- For all related LAB works of the lectures on scripting in bash shell, you shall make script files to put your commands and script codes in.
- You shall inform the instructor about your project group members by the next week.

What about TO DO parts?

- *TO DO parts are not graded. However, it is advised that you do.*

- There was a “TO DO” slide in the previous lecture material, Lecture5 presentation. Did you handle it?
  - The two sections of the textbook, “variable expansion modifiers” and “variable expansion of substrings” should be read and Table 13.18 and Table 13.19 should be examined at home.

Reading user input

- Use built-in command “read”
- While scripting, remember the following:
    - #!/bin bash
        - #! symbol in the first line of a script is sometimes called as **magic number**
- While scripting, also remember the following:
    - “export” command is used to make a variable global.
- read command is explained in Table 14.1.

The built-in “read” command

- read is mainly used to read input from the terminal or a file.
- read is mainly used to take a line of input.
  - read answer
    - reads a line from stdin and assigns it to the variable answer
  - read first last
    - reads a line from stdin to the first whitespace or newline, putting the first word into the variable first and the rest into the variable last
  - read
    - reads a line and assigns it to the built-in variable REPLY
  - read -a arrayName
    - reads a list of words into the array called arrayName

Arithmetic with variables

- declare -i variableName
  - declares variableName as integer
  - If any string value is attempted to be assigned, bash assigns 0 to the variable.
- If the variable has not been declared as integer, the built-in "let" command allows arithmetic operations.
  - $ i=5
  - OR
  - $ let i=5
  - $ let i=i+1
  - $ let "i = i +2" # quotes are needed for whitespace characters
  - $ ((i=i+3)) # double parentheses can replace let command
- **Note**: Bash only supports integer arithmetic, but the awk or bc utilities may be used for floating-point arithmetic.

Positional parameters and command-line arguments

- Data can be passed into a script via command line.
- Each word following the script name is an argument.
- Positional parameters:
  - The first argument is $1, the second is $2, and so on.
  - $0 is the reference to the name of the script.
  - $# holds the value of the number of the individual positional parameters (arguments).
  - $* lists all arguments.
- set command
  - set command with arguments resets the positional parameters. Once reset, the old argument list is lost.
    - set Ilker Kaya CE350

Some examples

- To understand the use of special :? modifier with arguments
    - **Example 14.13** shall be dissected in class.
- To understand the difference between $* and $@
    - **Example 14.14** shall be dissected in class.

Conditionals

- The exit status of 0 indicates success or TRUE, any exit status of nonzero indicates failure or FALSE.
- The status variable, ?, contains a numeric value representing the exit status.
    - $ echo $?
        - the output may be 0 or any numeric value according to the previous command's exiting state.

The built-in test and let commands

- test command or [ ] or [[ ]]
  - $ test kaya = ilker
  - $ test kaya==ilker
  - $ [kaya = ilker]
  - $ [kaya==ilker]
  - **[[is a keyword.**
- let command or (( ))
  - $ let x=2
  - $ let “x = x + 1”
  - $ ((x > 2)) # the expression is evaluated and an exit
    # status of 0 is returned.

File Testing

- Dissect Table 14.5
  - [-d $myFile]
  - [-f $myFile]
  - [-r $myFile –a –w $myFile]
  - OR
  - [[-d $myFile]]
  - [[-f $myFile]]
  - [[-r $myFile && –w $myFile]]

  ...

The if command

- The simplest form of conditional is “if”.
- if → then → fi
- echo “Are you there (y/n) ?”
- read replied
- if [“$replied” = Y -o “$replied” = y]
- then
    - echo “Okay then.”
- fi

The if/else command

- if → then → else → fi
- echo “Are you there (y/n) ?”
- read replied
- if [“$replied” = Y -o “$replied” = y]
- then
    - echo “Okay then.”
- else
    - echo “You are there since you replied !”
- fi

The if/elif/else command

- if → then → elif → then → else → fi
- echo “Are you there (y/n) ?”
- read replied
- if [“$replied” = Y -o “$replied” = y]
- then
    - echo “Okay then.”
- elif [“$replied” = N -o “$replied” = n]
- then
    - echo “You are there since you replied !”
- else
    - echo “Are you okay then?”
- fi

The null command

- The null command, **:**, refers to “do-nothing” and it returns an exit status of 0.
- echo “Are you there (y/n) ?”
- read replied
- if [“$replied” = Y -o “$replied” = y]
- then
    - **:** # do nothing
- else
    - echo “Why not telling the truth ?”
- fi

The case command

- A multiway branching command to be used as an alternative to if/elif commands.
- case variable in
- value1)
    - command(s)
    - ;;
- value2)
    - command(s)
    - ;;
- *)
    - command(s)
    - ;;
- esac

“case” example

- echo “Are you there (y/n) ?”
- read replied
- case “$replied” in
- [Yy]??)
    - echo “Okay then.”
    - ;;
- [Nn]?)
    - echo “You are there since you replied !”
    - ;;
- *)
    - echo “Are you okay then?”
- esac

Loops

- Bash has three kinds of loops:
  - **for**
  - **while**
  - **until**

The for command

- “for” is a keyword, which is a subunit of a command construct.
  - http://tldp.org/LDP/abs/html/internal.html#KEYWORDREF
- for anyVariable in aList
- do
  - command(s)
- done

A for example

- for items in Kaya Oguz Ilker Korkmaz CE350
- do
    - echo “the content of this item is: $items”
- done
- echo “out of the above loop”

The while command

- while command
- do
    - command(s)
- done

A while example

- i=1
- while (($i < 4)) # OR while [$i -lt 4]
- do
    - echo “in iteration $i”
    - let i+=1
- done
- echo “out of the above loop”

The until command

- until command
- do
    - command(s)
- done

An until example

- **until who | grep ilker**
- **do**
    - **sleep 5**
- **done**
- **echo “out of the above loop”**
- **# comments on the above script:**
    - The **until loop** tests the **exit status** of the last command in the pipeline, **grep**. The **who** command lists who is logged on the machine and **pipe**s its output to grep. If the grep command finds user ilker it returns a **0** exit status (**success**), any nonzero status (failure) otherwise.
    - If user ilker has not logged on, the body of the loop is entered and the program **sleep**s for 5 seconds.
    - When ilker logs on, the exit status of grep will be 0 and **control** will go to the statement following the **done** keyword.

The select command

- select variableName in wordList
- do
    - command(s)
- done
- # select loop iterates for each element in the wordList

- “select” is generally used with “case” to handle the menus

- Because the select command is a looping command, it is important to remember to use either the break command to get out of the loop, or exit command to exit the script.

A select-case example

- #!/bin/bash
- # Example 14.43 of the textbook.
- PS3="Please choose one of the three boys : "
- **select** choice in tom dan guy
- **do**
    - case "$choice" in
    - tom)
        - echo Tom is a cool dude!
        - **break**;; # break out of the select loop
    - dan | guy)
        - echo Dan and Guy are both wonderful.
        - **break**;;
    - *)
        - echo "$REPLY is not one of your choices" 1>&2
        - echo "Try again.“
        - ;;
    - esac
- **done**

continue vs break

- continue and break commands are used with similar objectives as they have in C programming language.
- an alternative:
  - continue 2
  - continue with a numeric argument makes the control be given from the inner loop to an outer loop.

Running loops in the background

- #!/bin/bash
- # Example 14.53 of the textbook.
- # memo is just a filename here
- # mail is a command
- **for** person in bob jim joe sam
- do
- mail $person < memo
- done **&**
- # In the body of the loop, each person is sent the contents of the memo file.
- # The **ampersand** at the end of the done keyword causes the loop to be #executed in the **background**. The program will continue to run while the loop is #executing.

Redirecting the output of a loop to a file

```
#!/bin/bash
# a snippet from Example 14.51 of the textbook.
cat $1 | while read line      # Input is coming from file provided at command line
do
    ((count == 1)) && echo "Processing file $1..." > /dev/tty
    echo -e "$count\t$line“
    let count+=1
done > tmp$$                  # Output is going to a temporary file
mv tmp$$ $1


# The read command returns a 0 exit status if it is successful in reading input
#and 1 if it fails.
# If the value of count is 1, the echo command is executed and its output is sent
#to /dev/tty , the screen.
# The output of this entire loop, each line of the file in $1, is redirected to the file
# tmp$$, with the exception of the first line of the file, which is redirected to the
#terminal, /dev/tty.
# $$ expands to the PID number of the current shell. By appending this number
#to the filename, the filename is made unique.
```

Functions

- A function is a name for a command or group of commands.
- Functions used in user scripts are executed in context of the current shell. A child process is not spawned as it is when running an executable program such as ls.
- Because the function is executed within the current shell, the variables will be known to both the function and the shell. Any changes made to your environment in the function will also be made to the shell.
- The local variables in a function are private to that function and will disappear after the function exits. (built-in **local** command)
- Functions may even be stored in another file to be loaded into the scripts when they are called to be used. (**source** command or dot (**.**))

Function formats

- function functionName { command ; command(s); }
- **OR**
- function functionName {
    - command(s)
- }
- **OR # similar to C style:**
- functionName () { command ; command(s); }
- **OR**
- functionName () {
    - command(s)
- }
- The keyword “function” requires the function name. The spaces surrounding the first curly brace are also required.
- export -f functionName # may be exported for subshells

Function arguments

- Arguments can be passed to functions via positional parameters. The positional parameters are private to the function and will not affect any positional parameters used outside the function.

- function Usage { echo “error: $*” 2>&1; exit 1; }
- a sample use may be as following:
    - if (($# != 1))
    - then
        Usage “$0 requires an argument”
    - fi

The return command

- The **return** command can be used to exit the function and return the control to the program at the place where the function was invoked.
- The return value of a function is actually the value of the exit status of the last command in the script, unless a specific argument to the return command is given. The argument of return must be an integer between 0 and 255.

A local and a return example

```
#!/bin/bash
# Example 14.57 of the textbook
increment ()  {
    local sum    # sum is known only in this function
    let "sum=$1 + 1"
    return $sum    # Return the value of sum to the script
}
echo  -n "The sum is "
increment 5    # Call function increment; pass 5 as a
               # parameter; 5 becomes $1 for the increment
               # function

echo $?        # The return value is stored in $?
echo  $sum     # The variable "sum" is not known here
```

Using command substitution for functions

```
#!/bin/bash
# Example 14.58 of the textbook
function square {
  local sq     # sq is local to the function
  let "sq=$1 * $1"
  echo "Number to be squared is $1."
  echo "The result is $sq "
}
echo "Give me a number to square. "
read number
value_returned=$(square $number)   # Command substitution
echo   "$value_returned"
# value_returned contains the output of the function, both of its echo statements.
```

Some extra:

- **dialog** command is used for interacting with the users through the simple graphical interfaces.
- The projects of previous semester included the “dialog” use.

TO DO

- http://tldp.org/LDP/abs/html/tests.html
  - Topics 7.1, 7.2, 7.3, 7.4
- http://tldp.org/LDP/abs/html/operations.html
  - Topics 8.1, 8.2, 8.3, 8.4
- http://tldp.org/LDP/abs/html/randomvar.html
  - Example 9.11
- http://tldp.org/LDP/abs/html/loops.html
  - Topics 11.1, 11.2, 11.3, 11.4
- http://tldp.org/LDP/abs/html/functions.html
  - Topic 24.2