LINUX USER GUIDE (V)

Working on the UNIX command line, part IV:

Unix shell interpreters, part II – scripting BASH

Cristian Vidrașcu

`cristian.vidrascu@info.uaic.ro`

March, 2024

Introduction 3

Bash scripting language features **5**

Scripts – shell procedures 6

Execution of shell procedures 7

Demo: a “Helloworld” script 9

Other features 10

Shell parameters **11**

Definition (creation) and use (substitution) 12

The assignment instruction 13

Other forms of substitution 14

Positional parameters and other special parameters 18

Useful builtin commands 19

Demo: a “PrintArgs” script 21

Arithmetic and conditional expressions **22**

Commands for arithmetic calculations 23

Commands for conditional expressions 25

Alternative and repetitive control structures **27**

The if builtin 28

The case builtin 29

The while builtin 30

The until builtin 31

The for builtin 32

The select builtin 33

Several useful builtins. 34

Shell functions **35**

Defining (declaring) and calling (executing) . . . 36
Examples of functions . . . 37

Bibliographical references **38**

Overview

Introduction

Bash scripting language features
- Scripts – shell procedures
- Execution of shell procedures
- *Demo*: a “Helloworld” script
- Other features

Shell parameters
- Definition (creation) and use (substitution)
- The assignment instruction
- Other forms of substitution
- Positional parameters and other special parameters
- Useful builtin commands
- *Demo*: a “PrintArgs” script

Arithmetic and conditional expressions
- Commands for arithmetic calculations
- Commands for conditional expressions

Alternative and repetitive control structures
- The if builtin
- The case builtin
- The while builtin
- The until builtin
- The for builtin
- The select builtin
- Several useful builtins

Shell functions
- Defining (declaring) and calling (executing)
- Examples of functions

Bibliographical references

Introduction

In the first part of this presentation we discussed about:

- Basic commands
  - How to execute basic commands
  - Execution of basic commands in the background
  - I/O redirects
- Compound commands
  - Pipelines
  - Sequential execution of several commands
  - Parallel, unchained execution of several commands
  - Conditional execution of several commands
  - Extended syntax: command lists
- Specifying file names
  - Specifying individual files
  - Patterns for multiple file specification
- Shell configuration files
  - Initializing the shell’s interactive sessions
  - The history of executed commands

Now we will continue the presentation with the topics summarized on the previous slide.

Introduction (cont.)

On a `UNIX` system, the command interpreter is an executable program that provides two basic functionalities:

- it takes commands entered by the user, interprets them, and executes them, thus realizing the interface between the user and the operating system;
- provides programming facilities in a specific command language, with which *scripts* can be written, *i.e.* text files containing sequences of `UNIX` commands.
  (*Important*: a scripting language allows the automation of work on the command line in that shell.)

I remind you that in the operating systems of the `UNIX` family we have several shells at our disposal: `sh` (*Bourne SHell*), `bash` (*Bourne Again SHell*), `csh` (*C SHell*), `ksh` (*Korn SHell*), `ash`, `zsh`, etc.

Regarding the second functionality above (*i.e.*, that of a scripting language), the shells available in `UNIX` possess all the facilities specific to any high-level programming language: variables, assignment statements, structured control statements – alternative and repetitive (like `if`, `case`, `while`, `for`, etc.), procedures and functions, call parameters, etc. The syntax of these facilities is specific to each shell, thus being a feature that differentiates `UNIX` shells from one another.

We will present all these facilities next, with explicit references to the syntax used by the scripting language provided by the command interpreter `bash`.

Outline

Introduction

Bash scripting language features
- Scripts – shell procedures
- Execution of shell procedures
- *Demo*: a “Helloworld” script
- Other features

Shell parameters
- Definition (creation) and use (substitution)
- The assignment instruction
- Other forms of substitution
- Positional parameters and other special parameters
- Useful builtin commands
- *Demo*: a “PrintArgs” script

Arithmetic and conditional expressions
- Commands for arithmetic calculations
- Commands for conditional expressions

Alternative and repetitive control structures
- The if builtin
- The case builtin
- The while builtin
- The until builtin
- The for builtin
- The select builtin
- Several useful builtins

Shell functions
- Defining (declaring) and calling (executing)
- Examples of functions

Bibliographical references

Scripts – shell procedures

A **shell procedure**, also called a script, is basically a text file containing sequences of simple and/or compound `UNIX` commands.
Note: also in `MS-DOS` or `Windows` we have a similar concept – the batch files `*.bat` .

To indicate a comment in a script, use the '#' character, followed by some text (up to the end of that line of text).

Accessing in the body of a procedure (*i.e.*, in that script file) its call arguments is done with the help of some special variables (which can only be read, not modified), having the following names: `$1` , `$2, ... , $9, ${10}, ${11}, ...`

* * *

Calling a script means invoking its name, using any of the syntactic forms allowed for external commands, already presented in the first part: the call forms of a simple command in foreground or in background, the call form of a pipeline and the other forms of specifying compound commands respectively.

Execution of shell procedures

Thus, to launch a script running in the foreground, one can use any of the three forms of call already mentioned, with the following particularities of execution:

```
UNIX> script_name [options] [arguments] [I/O redirects]
```

Outcome: a new process is created that non-interactively runs the shell specified on the first line of the script, by the syntax: `#!`*shell_name* (or the current shell if no shell is specified on the first line), and it will execute the commands in that file line by line (as if they were entered at its prompt).

```
UNIX> shell_name script_name [parameters and I/O redirects]
```

Outcome: as above, the newly created process runs the shell specified at the prompt.

```
UNIX> source script_name [parameters and I/O redirects]
```

or `UNIX> .  script_name [parameters and I/O redirects]`

Outcome: no new shell process is created, but the current shell process itself will execute the commands in that file line by line.

7 / 38

Execution of shell procedures (cont.)

And to launch a script running in the background, add the '&' character to the end of any of the three previously specified launch forms:

```
UNIX> script_name [parameters and I/O redirects] &

UNIX> shell_name script_name [parameters and I/O redirects] &

UNIX> source script_name [parameters and I/O redirects] &
```

or `UNIX> .  script_name [parameters and I/O redirects] &`

Outcome: the script is executed in the manner described in the three forms of calls for foreground execution, only this time the current interpreter no longer waits for the execution of that script to finish, but immediately redisplays the prompt, thus giving the user the possibility to enter a new command to be executed **before** the script finishes executing (*i.e.*, the new command will run in parallel with that script).

8 / 38

Demo: a “Helloworld” script

Here is an example, named `Hello.sh`, which illustrates the execution through the three forms of call:

```
#!/bin/bash
echo -e Hello, $1! \\n "The list of active processes in the current session:"
ps -f
```

UNIX> `./Hello.sh world`

```
Hello, world!
 The list of active processes in the current session:
UID        PID  PPID  C STIME TTY          TIME CMD
vidrascu 24249 24248  0 13:08 pts/4    00:00:00 -bash
vidrascu 24286 24249  0 13:08 pts/4    00:00:00 /bin/bash ./Hello.sh world
vidrascu 24287 24286  0 13:08 pts/4    00:00:00 ps -f
```

Note: here, the execution permission must be set (*i.e.*, `chmod u+x Hello.sh`) and the name must be specified by absolute or relative path.

UNIX> `dash Hello.sh Cristian`

```
-e Hello, Cristian!
 The list of active processes in the current session:
UID        PID  PPID  C STIME TTY          TIME CMD
vidrascu 24249 24248  0 13:08 pts/4    00:00:00 -bash
vidrascu 25002 24249  0 13:20 pts/4    00:00:00 dash Hello.sh Cristian
vidrascu 25003 25002  0 13:20 pts/4    00:00:00 ps -f
```

UNIX> `source Hello.sh`

```
Hello, !
 The list of active processes in the current session:
UID        PID  PPID  C STIME TTY          TIME CMD
vidrascu 24249 24248  0 13:08 pts/4    00:00:00 -bash
vidrascu 25082 24249  0 13:21 pts/4    00:00:00 ps -f
```

Other features

In the following sections we will introduce several facilities of the scripting language provided by the `bash` shell, such as:

- Shell variables
- Arithmetic and logical expressions
- Alternative and repetitive control structures
- Shell functions

Disclaimer: the presentation of these facilities will be at a simplified level. For a full description of all these facilities I recommend studying the `bash` shell documentation ([3]).

Outline

Introduction

Bash scripting language features
- Scripts – shell procedures
- Execution of shell procedures
- *Demo*: a “Helloworld” script
- Other features

Shell parameters
- Definition (creation) and use (substitution)
- The assignment instruction
- Other forms of substitution
- Positional parameters and other special parameters
- Useful builtin commands
- *Demo*: a “PrintArgs” script

Arithmetic and conditional expressions
- Commands for arithmetic calculations
- Commands for conditional expressions

Alternative and repetitive control structures
- The if builtin
- The case builtin
- The while builtin
- The until builtin
- The for builtin
- The select builtin
- Several useful builtins

Shell functions
- Defining (declaring) and calling (executing)
- Examples of functions

Bibliographical references

Definition (creation) and use (substitution)

A common feature of all command interpreters available on `UNIX` systems is the use of *variables*, with the caveat that variables have, by default, *string* values. They are also called *shell parameters*.

Variables are kept in a memory area of the shell process where they are defined (*i.e.*, created), as `name=value` pairs.

A variable is defined (*i.e.*, created) at the time of execution of the first *assignment statement* that involves it (*i.e.*, when it appears to the left of the attribution sign) or of the first `read` command that specifies it as an argument.

Remark: shell variables do not need to be declared in advance (*i.e.*, at the beginning of the procedure or function in which they are used), as in other high-level programming languages (*e.g.*, C/C++).

The *reference to the value of a variable* (*i.e.*, when we need the value of the variable in an expression) is made by its name preceded by the symbol '`$`', the effect being the substitution of the variable’s name by its value in the expression in which it appears.

Remark: also the command interpreters in `MS-DOS` or `Windows` have a similar concept of variables.

The assignment instruction

The *assignment instruction* is the internal command with the following syntax:
UNIX> *var*=[*expr*]
where *var* is an identifier (*i.e.*, a name) of a variable, and *expr* is an expression that must evaluate to a string (may include the empty string).
Warning: the '=' character must not be preceded or followed by white spaces!

Some examples of assignments and expressions using variable values:

```
UNIX> v=123                        # The variable v receives the value "123"
UNIX> echo $v                      # The text "123" is displayed on the screen
UNIX> cat xyv                      # The contents of the xyv file are displayed
UNIX> cat xy$v                     # The contents of the xy123 file are displayed
UNIX> v=abc${v}xyz                 # The variable v receives the value "abc123xyz"
UNIX> set                          # The list of defined variables is displayed
UNIX> unset v  or UNIX> v=         # v is "destroyed" (i.e., receives "empty string" as value)
```

Remark: the syntax ${*var*}*sufix* is used to indicate the desired substitution when the variable name is immediately followed, in that expression, by other characters that can be part of an identifier.

Other forms of substitution

Other forms of substitutions / expressions involving variables :

- *Use default values*: ${*var*:-*string*}
  Effect: the result of the expression is the value of the variable *var*, if it is defined, otherwise it is the value of *string* (and if *string* is missing, then a standard error message is displayed saying that the variable is undefined).
- *Assign default values*: ${*var*:=*string*}
  Effect: the expression's result is the value of the variable *var*, after it is eventually assigned the value *string* (assignment takes place only if *var* was undefined).
- *Display error if null or unset*: ${*var*:?*string*}
  Effect: the result of the expression is the value of the variable *var*, if it is defined, otherwise *string* is displayed (or a standard error message, if *string* missing).
- *Use alternate value*: ${*var*:+*string*}
  Effect: if *var* is already defined, then it is assigned the value *string*, otherwise it remains without value. Thus, the assignment takes place only if *var* was already defined.

Other forms of substitution (cont.)

Other forms of substitutions / expressions involving variables (cont.) :

- *Parameter length*: an expression of the form `${#var}`
  The effect is to be substituted with the length of the word / value of variable *var*.
- *Substring expansion*: `${var:start:length}` or `${var:start}`
  The effect is to be substituted with the subword, from the value of the variable *var*, starting from the position *start* and of length *length* (respectively, until the end of the word, in the case of the second form).

Here are some examples with such expressions / substitutions:

```
UNIX> word=12345                  # The variable word receives the value "12345"
UNIX> echo $word                  # The text "12345" is displayed on the screen
UNIX> echo ${#word}               # The text "5" is displayed on the screen
UNIX> echo ${word:0:2}            # The text "12" is displayed on the screen
UNIX> echo ${word:1:2}            # The text "23" is displayed on the screen
UNIX> echo ${word:2}              # The text "345" is displayed on the screen
```

Other forms of substitution (cont.)

- *Remove matching prefix pattern*: `${var#word}` or `${var##word}`
  Effect: will be substituted with the value of *var*, from which the shortest, resp. longest, prefix of it, equal to the value of the expression *word*, is removed.
- *Remove matching suffix pattern*: `${var%word}` or `${var%%word}`
  Effect: will be substituted with the value of *var*, from which the shortest, resp. longest, suffix of it, equal to the value of the expression *word*, is removed.

Here are some examples with such expressions / substitutions:

```
UNIX> v=AbcAbcAbcDEF              # The variable v is initialized, with the value "AbcAbcAbcDEF"
UNIX> echo ${v#A*c}               # The text "AbcAbcDEF" is displayed on the screen
UNIX> echo ${v##A*c}              # The text "DEF" is displayed on the screen
UNIX> fisier=/thor/profs/vidrascu/subdir/sursa.c    # The variable fisier is initialized
UNIX> echo ${fisier#/thor/profs/vidrascu/}          # The text "subdir/sursa.c" is displayed
UNIX> echo ${fisier##*/}          # The text "sursa.c" is displayed on the screen
UNIX> echo ${fisier%.c}.txt       # The text "/thor/profs/vidrascu/subdir/sursa.txt" is displayed
```

We can also use the external commands `dirname` and `basename` ([8]), like this:

```
UNIX> dirname $fisier             # The text "/thor/profs/vidrascu/subdir" is displayed on the screen
UNIX> basename $fisier .c         # The text "sursa" is displayed on the screen
```

Other forms of substitution (cont.)

- *Command substitution*: a special substitution is the expression `$(command)` or equivalent, `` `command` ``
  The effect is to be substituted, on the command line or in the context in which it appears, with the text displayed on the standard normal output by the execution of the specified command; this is executed in a *subshell* (*i.e.*, a child process of the current shell instance is created, which will run another instance of that shell).
- *Arithmetic expansion*: another special substitution is `$((expression))`
  The effect is to be substituted with the value of that arithmetic expression.

Word splitting: After interpreting the command line (*i.e.*, after evaluating the previously described variable, command, and arithmetic substitutions), the current shell instance "parses" the result of these substitutions, dividing it into words using as separators the characters <space>, <tab> and <newline>.

Pathname expansion: after splitting into words, the template substitutions for multiple file specification, described in the first part of this presentation, are performed.

Positional parameters and other special parameters

There are a number of variables that are dynamically modified by the current shell process, during command execution, in order to keep their meaning:

- `$1,$2,...,$9,${10},${11},...` Meaning: The positional parameters with which the current process was called (*i.e.*, parameters in the calling line in the case of a script).
- `$0` Meaning: the name of the current process (*i.e.*, of the script in which it is referenced).
- `$#` Meaning: number of positional parameters in the calling line (without the $0 argument).
- `$*` Meaning: list of positional parameters in the calling line (without the $0 argument).
- `$@` Meaning: list of positional parameters in the calling line (without the $0 argument).
  Remark: the difference between `$@` and `$*` occurs when they are used between quotes:
  on substitution `"$*"` produces a single word containing all parameters in the command line, while `"$@"` produces one word for each parameter in the command line.

- `$$` Meaning: the PID of the current shell process (*i.e.*, the shell instance executing that script).
- `$?` Meaning: The exit code returned by the last pipeline executed in the foreground.
- `$!` Meaning: the PID of the last process executed in the background.
- `$-` Meaning: the attributes with which the respective shell process was launched.
  Remark: these attributes (*i.e.*, execution options) can be handled with the builtin command `set`.

The `bash` shell also has a number of *predefined* environment variables (via the initialization files):
`$HOME` , `$USER` , `$LOGNAME` , `$SHELL` , `$MAIL` , `$PS1` , `$PS2` , `$TERM` , `$PATH` , `$CDPATH` , `$IFS` , etc.

Useful builtin commands

Here are some internal commands of the `bash` interpreter, useful for working with variables:

- *Read* command:
  `UNIX> read var [var2 var3 ...]`
  Has the effect of reading values from standard input and assigning them to the specified variables.
  Example: `UNIX> read -p "Input the number n:" n`
- *Read-only declaration* command:
  `UNIX> readonly var [var2 var3 ...]`
  The specified variables are declared as read-only (*i.e.*, they can no longer be modified after the execution of the command, but remain with the values they had when this command was executed).
- *Export* command:
  `UNIX> export var [var2 var3 ...]`
  It has the effect of "exporting" the specified variables to all child processes of the respective shell process (usually, variables are not "visible" in child processes, they are local to the respective shell process, being kept in its memory). We can also use the combination:
  `UNIX> export var=valoare [var2=valoare2 ...]`
  It has the effect of assigning and exporting the variable in a single command.

  In `UNIX` terminology, an exported variable is called an *environment variable*.

Useful builtin commands (cont.)

- The *shift* command, with the syntax: `shift [n]` , where *n* is 1, in case it is missing. It has the effect of "shifting" to the left by *n* positions all positional parameters (*e.g.*, for $n = 1$, in `$1` the value of `$2` is copied, in `$2` the value of `$3` is copied, and so on...), and the variables `$#`, `$@` and `$*` updates accordingly. It is useful, for example, when we want to sequentially process the call arguments of the respective procedure (see the example [FirstScript], iii), available here).
- The *eval* command, with the syntax: `eval parameters` . Effect: evaluates the specified parameters and executes the result of the evaluation (as if entered by the user at the prompt).
  An example: `eval newvar=\$$varname` . Effect: the value of the variable `varname` is considered as the variable identifier, and the value of the latter is assigned to the variable `newvar`.
  See also the example [FirstScript], iv)+v) in the lab support, available here).
- The *set* command is used to set (or unset) the attributes of the current shell.
  An example: `set -o noexec` or `set -n` . Effect: read commands but do not execute them.
  This may be used to check a shell script for syntax errors.
  Another example: `set -o xtrace` or `set -x` . Effect: for each command line entered by the user, it displays the result of interpreting that line, before actually executing it.
  This is useful for debugging purposes, to see exactly what the shell will execute after reading and interpreting that command line.
  Advice: add the `set -x` command to scripts to help you with writing and debugging scripts; and then, when you've reached the desired form of the script you're working on, you can remove it.
  For more details, see [Example #1 of a script with errors], available here.

Demo: a “PrintArgs” script

An example, named `PrintArgs.sh`, that illustrates positional parameter values through its execution:

```
#!/bin/bash
echo Positional parameters are: '$1'=$1, '$2'=$2, '$3'=$3, '$4'=$4, ...
echo The parameter '$0'=$0 , the parameter '$@'=$@ , and the parameter '$#'=$#
ps -o user,pid,ppid,cmd
```

```
UNIX> ./PrintArgs.sh hello world
```

```
Positional parameters are: $1=hello, $2=world, $3=, $4=, ...
The parameter $0=./PrintArgs.sh , the parameter $@=hello world , and the parameter $#=2
USER        PID  PPID CMD
vidrascu 29472 29471 -bash
vidrascu 30212 29472 /bin/bash ./PrintArgs.sh hello world
vidrascu 30214 30212 ps -o user,pid,ppid,cmd
```

```
UNIX> dash PrintArgs.sh val1 val2 val3
```

```
Positional parameters are: $1=val1, $2=val2, $3=val3, $4=, ...
The parameter $0=PrintArgs.sh , the parameter $@=val1 val2 val3 , and the parameter $#=3
USER        PID  PPID CMD
vidrascu 29472 29471 -bash
vidrascu 30351 29472 dash PrintArgs.sh val1 val2 val3
vidrascu 30352 30351 ps -o user,pid,ppid,cmd
```

```
UNIX> source PrintArgs.sh p1 p2 p3 p4 p5 p6
```

```
Positional parameters are: $1=p1, $2=p2, $3=p3, $4=p4, ...
The parameter $0=-bash , the parameter $@=p1 p2 p3 p4 p5 p6 , and the parameter $#=6
USER        PID  PPID CMD
vidrascu 29472 29471 -bash
vidrascu 30409 29472 ps -o user,pid,ppid,cmd
```

Outline

Introduction

Bash scripting language features
- Scripts – shell procedures
- Execution of shell procedures
- *Demo*: a “Helloworld” script
- Other features

Shell parameters
- Definition (creation) and use (substitution)
- The assignment instruction
- Other forms of substitution
- Positional parameters and other special parameters
- Useful builtin commands
- *Demo*: a “PrintArgs” script

Arithmetic and conditional expressions
- Commands for arithmetic calculations
- Commands for conditional expressions

Alternative and repetitive control structures
- The if builtin
- The case builtin
- The while builtin
- The until builtin
- The for builtin
- The select builtin
- Several useful builtins

Shell functions
- Defining (declaring) and calling (executing)
- Examples of functions

Bibliographical references

Commands for arithmetic calculations

Arithmetic expressions can be calculated with the builtin `let`, with the external commands `expr` or `bc` ([6]), or with the *arithmetic expansion* substitutions. Examples:

```
UNIX> let v=v-1                # The variable v is decremented.
UNIX> let v+=10                # The value of v is increased by 10.
UNIX> let "v += 2 ** 3"        # The value of v is increased by 8 (i.e., 2 raised to the 3rd power).
```

See also the example [FirstScript], i)-v) in the lab support, available here.

```
UNIX> expr 1 + 2 \* 3          # The value of the expression is displayed, i.e. 7.
UNIX> v=`expr $v - 1`          # The variable v is decremented.
UNIX> v=$(expr $v + 10)        # The value of v is increased by 10.
```

Another possibility is to work with *integer variables*: the command `declare -i n` sets the attribute “with integer values” for the variable `n`.

Arithmetic expressions can then be written directly, without explicitly using command `let`. Examples:

```
UNIX> n=5*4        # The variable n is assigned the value 20 (i.e., 5 multiplied by 4).
UNIX> n=2**3       # The variable n is assigned the value 8 (i.e., 2 raised to the 3rd power).
```

We can also work with *vector variables*: the command `declare -a v` sets the attribute “array” for the variable `v`. The reference to an element of the vector is done by `v[i]`.
For illustration, see the example [Iterative math #4] in the lab support, available here.

Commands for arithmetic calculations (cont.)

The command bc ([6]) – a programming language for floating point calculations. Here are some calculation examples:

```
UNIX> echo 3 ^ 2 | bc                                    # The value 9 is displayed.
UNIX> echo 3/ 2 | bc                             # The integer value 1 is displayed.
UNIX> echo 3 /2 | bc -l          # It displays 1.500...000 (i.e., with 20 decimal places).
UNIX> echo "scale=4; 3/2" | bc         # It displays 1.5000 (i.e., with 4 decimal places).
UNIX> echo "scale=10; 4*a(1)" | bc -l     # It displays 3.1415926532, i.e. the number π,
                   with 10 decimal places ; a(1) is the call to the library function arctan(x) .
UNIX> echo "scale=5; sqrt(2)" | bc      # It displays 1.41421, i.e. √2 with 5 decimal places.
UNIX> echo "scale=2; v = 1; v += 3/2; v+10" | bc            # The value 12.50 is displayed.
```

See also the example [MyExpr] in the lab support, available here.

Arithmetic expansion using the syntax ((...)) and $((...)) . Some examples:

```
UNIX> a=$(( 4 + 5 ))                            # The variable a is assigned the value 9.
UNIX> (( a += 10 ))                                 # The value of a is increased by 10.
UNIX> (( b = a<45?11:22 ))                     # The variable b is assigned the value 11.
UNIX> echo $((0xFFFF))          # It displays 65535, i.e. the value of that hexadecimal number.
UNIX> echo $((4#1203))       # It displays 99, i.e. the value of the number 1203 written in base 4.
```

24 / 38

Commands for conditional expressions

The internal command to *evaluate a conditional expression* ([7]) has the following syntax:

test *expression*   or   [*expression*] ,

where the conditional expression *expression* can be of one of the following forms:

- Relational operators on strings:

```
— test -z string                                        # True if string is 0 in length.
— test -n string   or   test string              # True if string is not the empty string.
— test string_1 = string_2                           # True if the two strings are equal.
— test string_1 != string_2                      # True if the two strings are not equal.
— test string_1 < string_2                     # Comparison using lexicographic order.
— test string_1 > string_2                     # Comparison using lexicographic order.
```

- Relational operators between two numeric values:

```
— test val_1 -eq val_2                                # True if the two values are equal.
— test val_1 -ne val_2                            # True if the two values are not equal.
— test val_1 -gt val_2                        # Tests the inequality “strictly greater”.
— test val_1 -ge val_2                 # Tests the inequality “greater than or equal”.
— test val_1 -lt val_2                           # Tests the inequality “strictly less”.
— test val_1 -le val_2                  # Tests the inequality “less than or equal to”.
```

25 / 38

Commands for conditional expressions (cont.)

(cont.) Other conditional expressions usable as arguments of the internal command `test`:

- Tests on files: `test -opt file` , where the test option `-opt` can be:
  - `-e` : test for the existence of that file (of any type);
  - `-d` : tests if the file exists and is of type directory;
  - `-f` : tests if the file exists and is a regular file type;
  - `-p` : tests if the file exists and is of type fifo;
  - `-b` / `-c` : tests if the file exists and is of type device, in block / character mode;
  - `-r` / `-w` / `-x` : tests if the current user can read / modify / execute the file;
  - `-s` : tests if the file has non-empty content ; etc. (see `help test` and `man 1 test`)
- A logical expression (negation, conjunction, or disjunction of expressions):
  - `test ! expression_1` # The negation of the expression *expression_1*
  - `test expression_1 -a expression_2` # The conjunction of the two expressions
  - `test expression_1 -o expression_2` # The disjunction of the two expressions

  expression_1 and *expression_2* being expressions of any of the forms previously specified.

Remark: there is also the compound command `[[ expression ]]` available in the shell `bash`, with the same expressions used by `test`, but with some differences in execution (*e.g.*, *string_2* can be a regular expression and string relational operators will do pattern matching in this situation).

Outline

Introduction

Bash scripting language features
- Scripts – shell procedures
- Execution of shell procedures
- *Demo*: a “Helloworld” script
- Other features

Shell parameters
- Definition (creation) and use (substitution)
- The assignment instruction
- Other forms of substitution
- Positional parameters and other special parameters
- Useful builtin commands
- *Demo*: a “PrintArgs” script

Arithmetic and conditional expressions
- Commands for arithmetic calculations
- Commands for conditional expressions

Alternative and repetitive control structures
- The if builtin
- The case builtin
- The while builtin
- The until builtin
- The for builtin
- The select builtin
- Several useful builtins

Shell functions
- Defining (declaring) and calling (executing)
- Examples of functions

Bibliographical references

The if builtin

1) The *if builtin* has the following syntax:

```
if command_list_1 ; then command_list_2 ; [else command_list_3 ;] fi
```

Or, any of the three occurrences of the ’;’ character can be replaced with <newline>. E.g:

```
 if command_list_1
 then
   command_list_2
[ else
   command_list_3 ]
 fi
```

Semantics: the commands in `command_list_1` are executed first and, if the exit code of the last command in this list is 0 (*i.e.*, successful completion), then the commands in `command_list_2` are executed. Otherwise (but only if the `else` branch also exists), the commands in `command_list_3` are executed.

“Multiple IFs” – we can have multiple nested `if`s (abbreviated `elif`) :

```
if list_1 ; then list_2 ; [elif list_3 ; then list_4 ;]...[else list_N ;] fi
```

The case builtin

2) The *case builtin* has the following syntax:

```
case expression in
  value_list_1   ) command_list_1   ;;
  value_list_2   ) command_list_2   ;;
      ......          ......
  value_list_N-1 ) command_list_N-1 ;;
  value_list_N   ) command_list_N
esac
```

Semantics: `expression` is evaluated and each line is “scanned” looking for the first line that contains it: if the value of the expression is found in the list of values `value_list_1`, then `command_list_1` is executed and then the execution of `case` command ends. Otherwise, if the value of the expression is found in the list of values `value_list_2`, then `command_list_2` is executed and then the execution of `case` command ends. Otherwise, . . . and so on. . .

Remark: if “`;;&`” is used instead of “`;;`”, then the lists of commands associated with all lists of values containing the value of that expression will be executed (*i.e.*, the execution of the command `case` no longer ends after finding the first line containing the value of the expression).

29 / 38

The while builtin

3) The *while builtin* has the following syntax:

```
while command_list_1 ; do command_list_2 ; done
```

Either of the two occurrences of the ’;’ character can be replaced by <newline>. E.g:

```
while command_list_1
do
  command_list_2
done
```

Semantics: on each iteration of the loop, the commands in `command_list_1` are executed, and if the exit code of the last command in this list is 0 (*i.e.*, successful completion), then the commands from `command_list_2` are executed and the loop resumes. Otherwise, the execution of the `while` loop ends.

30 / 38

The until builtin

4) The *until builtin* has the following syntax:

```
until command_list_1 ; do command_list_2 ; done
```

Either of the two occurrences of the ';' character can be replaced by <newline>. E.g:

```
until command_list_1
do
  command_list_2
done
```

Semantics: on each iteration of the loop, the commands in `command_list_1` are executed, and if the exit code of the last command in this list is different from 0 (*i.e.*, terminate with failure), then the commands from `command_list_2` are executed and the loop resumes. Otherwise, the execution of the `until` loop ends.

The for builtin

5) The *for builtin* has the following syntax:

```
for variable [ in word_list ] ; do command_list ; done
```

Either of the two occurrences of the character ';' can be replaced by <newline>. E.g:

```
for variable [ in word_list ]
do
  command_list
done
```

Semantics: `word_list` describes a list of values that the `variable` takes successively, and for each such value the commands in `command_list` are executed.

Remark: this form of the `for` loop is used for unordered sets of values given by enumeration. But if we have an ordered set of values, we could specify it by the minimum value, the maximum value, and the increment step. For this purpose, the command `seq` can be used, as follows:

```
UNIX> for v in $(seq first increment last) ; do command_list ; done
```

Alternatively, the command `for ((` can also be used, with the following syntax:

```
for (( exp1; exp2; exp3 )); do command_list ; done
```

where `exp1`, `exp2` and `exp3` are arithmetic expressions, with the same meanings as in C/C++.

The select builtin

6) The *select builtin* has the following syntax:

```
select variable [ in word_list ] ; do command_list ; done
```

Either of the two occurrences of the character ';' can be replaced by <newline>. E.g:

```
select variable [ in word_list ]
do
  command_list
done
```

Semantics: is a combination of `for` and `case` – at each iteration the *`variable`* receives as value that word from the *`word_list`* which is selected through user interaction. The execution of the `select` loop is also terminated by user interaction (pressing `CTRL+D` keys, *i.e.* EOF).

Remark:
In both the `select` loop and the `for` loop, if the optional part `in word_list` is missing, then the value of the special variable `$@` is processed as the list of words.

Several useful builtins

Here are some other internal commands, useful both in scripts and on the `bash` shell command line:

- The `break` builtin, with syntax: `break [n]`, where *n* is 1, if missing. Effect: Breaks out of *n* nested `do-done` loops, with execution continuing with the next statement after `done`.
- The `continue` builtin, with syntax: `continue [n]`, where *n* is 1, if missing.
  Effect: in the case of $n = 1$ the current `do-done` loop is restarted (from the reinitialization step), respectively for $n > 1$ the effect is as if executing *n* times the command `continue 1`.
- The `exit` builtin, with syntax: `exit [code]`, where *code* is the value of the special variable `$?`, in case it is missing. Effect: terminates execution of the script (or shell instance) in which it is called, and its exit code will be the specified value.
- The `wait` builtin, with syntax: `wait [pid]`. Effect: suspends execution of the calling script (or shell instance), waiting for the process with the specified PID to finish.
- The `exec` builtin, with syntax: `exec command`. Effect: executes the specified command, without creating a new process for it. Thus, the shell that executes this command will be “replaced” with the process associated with the command (so it is not *reentrant*).
- The `trap` builtin, with syntax: `trap command event`. Effect: the specified command will be executed when that event occurs (*i.e.*, when that signal is received).
  Example: `UNIX> trap 'rm /tmp/ps$$ ; exit' 2`
  Note: 2 = *interrupt signal* (the signal generated by pressing the keys `CTRL+C`), 3 = *quit signal* (the signal generated by pressing the keys `CTRL+\`), etc.

Outline

Introduction

Bash scripting language features
- Scripts – shell procedures
- Execution of shell procedures
- *Demo*: a “Helloworld” script
- Other features

Shell parameters
- Definition (creation) and use (substitution)
- The assignment instruction
- Other forms of substitution
- Positional parameters and other special parameters
- Useful builtin commands
- *Demo*: a “PrintArgs” script

Arithmetic and conditional expressions
- Commands for arithmetic calculations
- Commands for conditional expressions

Alternative and repetitive control structures
- The if builtin
- The case builtin
- The while builtin
- The until builtin
- The for builtin
- The select builtin
- Several useful builtins

Shell functions
- Defining (declaring) and calling (executing)
- Examples of functions

Bibliographical references

Defining (declaring) and calling (executing)

A **shell function** is a name for a sequence of UNIX commands, analogous to shell procedures, except that a function is not written in a separate text file, as in their case, but is written (*i.e.*, *declared*) either in a shell procedure, or directly at the prompt of a shell instance, using the syntax:

```
function function_name () { command_list ; }
```

Semantics: the `function` builtin declares *function_name* to be a *variable of type function*, that is, an “alias” for the sequence of commands *command_list* .

Calling a function means *the execution of that sequence of commands* and it is done similarly as for any simple command, *i.e.* by its name, plus parameters and possible I/O redirects.

As with procedures, in the body of a function (*i.e.*, in *command_list*) we use the positional parameters `$1, $2, ... , $9, ${10}, ${11}, ...` to refer to the function’s call parameters, and by the special variables `$#`, respectively `$*` and `$@`, we mean the number, respectively the list, of all call parameters of that function.

Remarks: i) in the declaration of a function, either `function` or `()` can be omitted, but not both!
ii) nothing is ever written between the pair of parentheses `()`, even if we want to declare the function as having one or more arguments! iii) in conclusion, the concept of *function* from shells is NOT similar, syntactically and semantically, to that of a function in programming languages like C/C++.

Examples of functions

1) Here is an example of a function, declared and called directly on the shell command line:

```
UNIX> function my_listing () { echo "The directory listing: $1" ; \
> if test -d $1 ; then ls -lA $1 ; else echo "Error" ; fi }
UNIX> my_listing ~vidrascu/so/
```

Effect: the contents of the directory specified as the function argument will be displayed.

2) Here is another example of a function, declared and called inside a script:

```
#!/bin/bash
function count_params ()
{
  echo "Call with $# argument(s) : $*"
}
count_params "$*"   # The first call of the function.
count_params "$@"   # The second call of the function.
```

If we call this script with the following command line, we will get as output:

```
UNIX> ./script.sh a b c
```

```
Call with 1 argument(s) : a b c
Call with 3 argument(s) : a b c
```

Conclusion: the on-screen messages show us the difference between how the special variables `$*` and `$@` are evaluated when they are enclosed in quotes.

3) Other examples: see also [MyGccOrCat] and [Recursive math #1 & #2] in the lab support, available here.

Bibliographical references

Mandatory bibliography

[1] Chapter 2, §2.4 from the book "*Sisteme de operare – manual pentru ID*", by C. Vidrașcu, UAIC Publishing House, 2006. This is available as ebook at the address:

- `https://profs.info.uaic.ro/~vidrascu/SO/books/ManualID-SO.pdf`

[2] The online support lesson associated with this presentation:

- `https://profs.info.uaic.ro/~vidrascu/SO/support-lessons/bash/en/support_lab4.html`
- `https://profs.info.uaic.ro/~vidrascu/SO/support-lessons/bash/en/support_lab5.html`

Additional bibliography:

[3] The documentation of `bash` shell : `man 1 bash` and "GNU Bash manual"

[4] *Linux Documentation Project Guides* $\longrightarrow$ "Advanced Bash-Scripting Guide"

[5] The book "Bash Pocket Reference" (1st edition), by A.Robbins, O'Reilly Media Inc., 2010.

[6] `help let` , `man 1 expr` and `man 1 bc` / `man 1p bc`

[7] `help test` and `man 1 test`

[8] `man 1 basename` and `man 1 dirname`