Bash Scripts

Jose L. Muñoz, Oscar Esparza, Juanjo Alins, Jorge Mata
Telematics Engineering
Universitat Politècnica de Catalunya (UPC)

Outline

What is a Shell Script?

- Some of the files on a UNIX system are deemed *executable*.
- These can be thought of as *programs*
- Some of these programs are compiled from source code (such as C).
- These are sometimes known as *binary executables*.
- Other executables contain only text to be interpreted.
- These are called *scripts*.
- We have scripts for shells.
- Also, there are other different interpreters for scripts, such as `python` and `perl`.

Which Shell?

- There are a variety of UNIX shells to choose from.
- The original and most widely supported shell is called the *Bourne shell* (after S.R. Bourne).
- The binary is usually in /bin/sh.
- 95% of the world's shell scripts are created for use with the Bourne Shell.
- There are a number of Bourne shell derivatives, each offering a variety of extra features, including:
  - The Korn shell (after David Korn) (**ksh**) (not open/free).
  - The Bourne-again shell (**bash**) (open/free).
  - zsh.
- Such shells are supposed to be Bourne-compatible.

Shebang

- The first 2 chars of a script should be “#!”.
- This is called the **shebang**.
- The kernel executes the name that follows, with the file name of the script as a parameter.
- Example: a file called `find.sh` has this as the first line:

```
#! /bin/sh
```

- Then, kernel executes:

```
/bin/sh find.sh
```

- For bash scripts:

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

- If you make any typing mistake in the name of the interpreter, you will get an error message such as
  `bad interpreter: No such file or directory.`

Outline

Quoting I

- Quoting is used when defining variables.

```
$ MYVAR=Hello
```

- If we want to define a variable with a value that contains spaces or tabs we need to use quotes.

```
$ MYVAR='Hello World'
```

- Single quotes mean literal:

```
#!/bin/bash
MYVAR='Hello world'
MYVAR2='$MYVAR, How are you?'
echo $MYVAR2
```

Quoting II

	Quotes	Meaning
'	simple	The text between single quotes is treated as a literal (without modifications). In bash, we say that it is not *expanded*.
"	double	The text between double quotes is treated as a literal except for what follows to characters \, `and $.
`	reversed	The text between reverse quotes is interpreted as a command, which is executed and whose output is used as value. In bash, this is known as *command expansion*.

- Example double quotes:

```
#!/bin/bash
MYVAR='Hello world'
echo "$MYVAR, how are you?"
```

Quoting III

- Reverse quotes cause the quoted text to be interpreted as a shell command:

```
$ echo "Today is `date`"
Today is Tue Aug 24 18:48:08 CEST 2008
```

- Braces {} are used to separate variables:

```
$ MYVAR='Hello world'
$ echo "foo$MYVARbar"
foo
```

- We have to use braces to resolve the ambiguity:

```
$ echo foo${MYBAR}bar
fooHello worldbar
```

Outline

Positional & Special Params I

- Often, you will need that your script can process arguments when invoked.
- These arguments are called positional parameters: $1 to $N.
- $0 is the basename of the script as it was called.

```
#!/bin/bash
echo "$1"
```

- Execute:

```
$ ./my_script.sh Hello word
Hello
$ ./my_script.sh 'Hello word'
Hello word
```

Positional & Special Params II

- Another example:

```
#!/bin/bash
echo Script name is "$0"
echo First positional parameter $1
echo Second positional parameter $2
echo Third positional parameter  $3
echo The number of positional parameters is $#
echo The PID of the script is $$
```

- $# expands to the number of positional parameters.
- $$ expands to the PID of the bash that is executing the script,
- $@ expands to all the positional parameters (also $* does this).
  - $@ and $* both expand to all the positional parameters.
  - There are no differences between $* and $@,
  - But there is a difference between "$@" and "$*".
  - "$*" means always one single argument,
  - "$@" contains as many arguments.

Positional & Special Params III

- Example:

```
$ cat script1.sh
mkdir "$*"
$ cat script2.sh
mkdir "$@"
$ ./script1 dir1 dir2 ; ./script2 dir3 dir4
$ ls | grep dir
dir1 dir2
dir2
dir3
```

- ”$@” is a special token which means ”wrap each individual argument in quotes”.

Outline

Expansions

- Before executing your commands, bash checks whether there are any syntax elements in the command line that should be interpreted rather than taken literally.
- These are called expansions and we have several types.
- In processing order they are:
  - **Brace Expansion**: create multiple text combinations.
  - **Tilde Expansion**: expand useful pathnames home dir, working dir and previous working dir.
  - **Parameter Expansion**: how bash expands variables to their values.
  - **Command Substitution**: using the output of a command as an argument.
  - **Arithmetic Expansion**: how to use arithmetics.
  - **Process Substitution**: a way to write and read to and from a command.
  - **Filename Expansion**: a shorthand for specifying filenames matching patterns.

Brace Expansion

- Brace expansions are used to generate all possible combinations with the optional surrounding preambles and postscripts.
- The general syntax is:
  `[preamble]{X,Y[,...]}[postscript]`
- Examples:

```
$ echo a{b,c,d}e
abe ace ade
$ echo "a"{b,c,d}"e"
abe ace ade
$ echo "a{b,c,d}e"
a{b,c,d}e
$ mkdir $HOME/{bin,lib,doc}
$ echo {x,y}{1,2}
```

- The are also a couple of alternative syntaxes for brace expansions using two dots:

```
$ echo {5..12}
5 6 7 8 9 10 11 12
$ echo {c..k}
c d e f g h i j k
$ echo {1..10..2}
1 3 5 7 9
```

Tilde Expansion

- The tilde expansion is used to expand three specific pathnames:
  - Home directory: ~
  - Current working directory: ~+
  - Previous working directory: ~-
- Examples:

```
$ cd /
/$ cd /usr
/usr$ cd ~-
/$ cd ~
~$ cd /etc
/etc$ echo ~+
/etc
```

Parameter Expansion I

- Parameter expansion allows us to get the value of a parameter (variable).
- On expansion time, you can do extra processing with the parameter or its value.
- **${VAR:-string}** If the parameter VAR is not assigned, the expansion results in 'string'. Otherwise, the expansion returns the value of VAR. For example:

```
$ VAR1='Variable VAR1 defined'
$ echo ${VAR1:-Variable not defined}
Variable VAR1 defined
$ echo ${VAR2:-Variable not defined}
Variable not defined
```

Parameter Expansion II

- You can also use positional parameters. Example:

```
USER=${1:-joe}
```

- **${VAR:=string}** If the parameter VAR is not assigned, VAR is assigned to 'string' and the expansion returns the assigned value. Note. Positional and special parameters cannot be assigned this way.
- There are more parameter expansions described in our book.

Command Substitution

- Command substitution yields as result the standard output of the command executed.
- Syntaxes:
  `$(COMMAND)` or `` `COMMAND` ``
- We also can use command substitution together with parameter expansion.
- Example:

```
$ echo VAR=${VAR1:-Parameter not defined in `date`}
Parameter not defined in Thu Mar 10 15:17:10 CET 2011
```

Arithmetic Expansion

- `bash` is primary designed to manipulate text strings but it can also perform arithmetic calculations.
- Syntax is: ((...)) or $[...]
- Examples:

```
VAR=55              # Asign the value 55 to the variable VAR.
((VAR = VAR + 1))   # Adds one to the variable VAR (notice that we don't use $).
((++VAR))           # C style to add one to VAR (preincrease).
((VAR++))           # C style to add one to VAR (postincrease).
echo $[VAR * 22]    # Multiply VAR by 22
echo $((VAR * 22)) # Multiply VAR by 22
```

- We also can use the extern command `expr` to do arithmetic operations:

```
$ X=`expr 3 \* 2 + 7`
$ echo $X
13
```

- However, `expr` is a new process (less efficient).
- `bash` cannot handle floating point calculations.
- For this purpose you can use `bc`.

Process Substitution

- When you enclose several commands in parenthesis, the commands are actually run in a “subshell”.
- Since the outer shell is running only a “single command”, the output of a complete set of cmds can be redirected as a unit.
- Process substitution occurs when you use $<$ or $>$ with parenthesis:
  - cmdX $<$(cmds) redirects the output of cmds to the input of cmdX. For example:

```
$ cat <(ls -l)
```

  - cmdX $>$(cmds) redirects the output of cmdX to the input of cmds. For example:

```
$ (ps ; ls) >commands.out
```

- In detail, the previous command line uses a temporary named pipe, which bash creates, names and later deletes.

Filename Expansion

- A pattern is replaced by a list names of files sorted alphabetically. Possible patterns:

Format	Meaning
*	Any string of characters, including a null (empty) string.
?	Any unique character.
[*List*]	A unique character from the list. We can include range of characters separated by hyphens (-). If the first character of the list is ^ or !, this means any single character that it is not in the list.

- Examples:

```
$ ls *.txt
file.txt  doc.txt  tutorial.txt
$ ls [ft]*
file.txt  tutorial.txt
$ ls [a-h]*.txt
file.txt  doc.txt
$ ls *.??
script.sh file.id
```

- Filename expansions use glob patterns, which are a type of regular expression.
- Regular expressions are described next.

Outline

Introduction

- A regular expression is a special text string for describing a search pattern mainly for use in find and replace like operations.
- In Linux systems, we have to types of regular expressions: **glob patterns** and **regular expressions (regex)**.
- glob patterns are the oldest and simplest type of regular expressions used in Unix-like systems.
- They are used by commands related with the file system like `ls`, `rm`, `cp` etc.
- regex are more versatile regular expressions.
- regex are used by commands related to string manipulation and filtering like `grep` or `sed`.

Glob Patterns

- We have two sets of globs, basic and extended.
- The **basic globs** and their meaning are listed below:

Character	**Meaning**
?	Expands one character.
*	Expands zero or more characters (any character).
[ ]	Expands one of the characters inside [ ].

Extended globs

- To use the following expansions you have to have activated extended globbing.
- You can check this with:

```
$ shopt -s extglob
```

- The **extended globs** and their meaning are listed below:
  - ?(...|...) Expands **zero or one** occurrence of the items listed between the pipes
  - *(...|...) Expands **zero or more** occurrence of the items listed between the pipes
  - +(...|...) Expands **one or more** occurrence of the items listed between the pipes
  - @(...|...) Expands **any** occurrence of the items listed between the pipes
  - !(...|...) Expands anything **except** the items listed between the pipes

Examples of globs

- Examples:

```
$ touch 03.txt ; touch {aa,b,c}{00,01,02}.txt; ls
03.txt a00.txt  a01.txt  a02.txt  aa00.txt  aa01.txt  aa02.txt
b00.txt  b01.txt  b02.txt  c00.txt  c01.txt  c02.txt

$ ls *(a|b)??.txt
03.txt a00.txt  a01.txt  a02.txt  aa00.txt  aa01.txt  aa02.txt  b00.txt  b01.txt  b02.txt

$ ls ?(a)0*
03.txt  a00.txt  a01.txt  a02.txt

$ ls +(a|?0)*
a00.txt  a02.txt   aa01.txt  b00.txt  b02.txt  c01.txt
a01.txt  aa00.txt  aa02.txt  b01.txt  c00.txt  c02.txt

$ ls @(aa|*2)*
a02.txt  aa00.txt  aa01.txt  aa02.txt  b02.txt  c02.txt

$ ls !(a0*|aa0*)
03.txt b00.txt  b01.txt  b02.txt  c00.txt  c01.txt  c02.txt
```

Regular Expressions (regex)

- Regular Expresions (regex) provide three main features:
  - **Concatenation**. RE ab matches an input string of ab.
  - **Union**. RE $a|b$ matches an input string of a or an input string of b. It does not match ab.
  - **Closure**. RE a* matches the empty string, or an input string of a, or an input string of aa, etc.
- We have two sets of regex, basic and extended.

Basic Regex I

- The **basic regex** and their meaning are listed below:
  - `\` is used to escape special characters.
  - `.` matches any single character of the input line.
  - `^` This character does not match any character but represents the beginning of the input line. For example, `^A` is a regular expression matching the letter A at the beginning of a line.
  - $ This represents the end of the input line.
  - `[ ]` A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches a, b, or c. [a-z] specifies a range which matches any lowercase letter from a to z. These forms can be mixed: [abcx-z] matches a, b, c, x, y, or z, as does [a-cx-z].

Basic Regex II

- `[^ ]` Matches a single character that is not contained within the brackets.
- RE* A regular expression followed by * matches a string of zero or more strings that would match the RE. For example, A* matches A, AA, AAA, and so on. It also matches the null string (zero occurrences of A).
- `\( \)` are used for grouping. The RE inside the escaped parenthesis creates a group that can be referenced with `\1` through `\9` (up to nine groups can be created).
- `\{ \}` means the same as {m,n} (without backslash) does in Extended regex (see next Section).

- Example:

```
$ ps -u $USER | grep '^ [0-9][0-9][0-9]9'
```

Extended regex

- The **extended regex** and their meaning are listed below:
  - `RE+` A regular expression followed by + matches a string of one or more strings that would match the RE.
  - `RE?` A regular expression followed by ? matches a string of zero or one occurrences of strings that would match the RE.
  - `()` which are parenthesis without backslash, are used for grouping in extended regex.
  - `{}` is used to create unions. Examples:
    - `a{3}` is equivalent to regular expression aaa (exactly `a` three times).
    - `a{3,}` is equivalent to `aaaa*` (3 or more `a`).
    - `a{,3}` is equivalent to `|a|aa|aaa` (matches the empty string or `a` or `aa` or `aaa`).
    - `a{3,5}` is equivalent to regular expression `aaa|aaaa|aaaaa`.

Examples of regex

- `abc` matches any line of text containing abc.
- `.at` matches any three-character string ending with at, including hat, cat, and bat.
- `[hc]at` matches hat and cat.
- `[^b]at` matches all strings matched by .at except bat.
- `^[hc]at` matches hat and cat at the beginning of string.
- `[hc]at$` matches hat and cat at the end of string.
- `\[.\]` matches any single character surrounded by [], for example: [a] and [b].
- `^.$` matches any line containing exactly one character (the newline is not counted).
- `.* [a-z]+ .*` matches any line containing a word, consisting of lowercase alphabetic characters, delimited by at least one space on each side.

Outline

Introduction I

- `sed` is a stream editor.
- A stream editor is used to perform basic text transformations on an input stream (a file, or input from a pipeline).
- `sed` is very efficient because it only makes one pass over the input(s).
- `sed` maintains two data buffers (both are initially empty):
  - active pattern space.
  - auxiliary hold space.
- In normal operation:
  - `sed` reads in one line from the input stream.
  - Removes any trailing newline.
  - Places the line in the pattern space.
  - The pattern space is where text manipulations occur.

Introduction II

- A set of commands are applied to make the manipulations.
- Each command can have an address associated to it:
  - An address is a kind of condition code.
  - A command is only executed if the condition is verified.
- When the end of the line is reached the contents of pattern space are printed out to the output stream.
- The trailing newline is addded back if it was removed.
- Then the next cycle starts for the next input line.
- The pattern space is deleted between two cycles.
- The hold space, on the other hand, keeps its data between cycles.
- The hold space is initially empty, but there are commands for moving data between the pattern and hold spaces.

Substitute Command

- The substitute command changes all occurrences of the regular expression into a new value:

```
$ sed s/day/night/ <f1.txt >f2.txt
```

- If you have meta-characters in the command, quotes are necessary:

```
$ sed 's/day/night/'<f1.txt >f2.txt
```

- It's a good habit to use quotes.
- The substitute command has four parts:

```
s          Substitute command
/../../    Delimiter
day        Search Pattern (Regular Expression)
night      Replacement string
```

Global replacement

- If you tell `sed` to change a word, it will only change the first occurrence of the word on a line.
- You may want to make the change on every word on the line instead of the first.
- If you want it to make changes for every match, add a `g` after the last delimiter:

```
$ sed 's/one/ONE/g' < f.txt
ONE two three, ONE two three
four three two ONE
ONE hundred
```

Using the Matched String I

- Sometimes you want to search for a pattern and add some characters to the matched string.
- For this purpose, you can use ”&“:

```
$ sed 's/[a-z]*/(&)/' <old >new
```

- The above command surrounds a string of any length with parentheses.
- You can have any number of & in the replacement string:

```
$ echo "123 abc" | sed 's/[0-9]*/& &/'
123 123 abc
```

- But if we do this:

```
$ echo "abc 123" | sed 's/[0-9]*/& &/'
 abc 123
```

- First character matched is ZERO (replaced by a space in this example).

Using the Matched String II

- A better way to duplicate the number is to make sure it matches a number:

```
$ echo "abc 123" | sed 's/[0-9][0-9]*/& &/'
abc 123 123
```

- You can use also extended regular expressions (sign +).
- To enable them, you have to use the `-r` option of `sed`:

```
$ echo "abc 123" | sed -r 's/[0-9]+/& &/'
abc 123 123
```

Keep Part of the Pattern I

- The escaped parentheses remember a substring of the characters matched by the regular expression.
- `\1` is the first remembered pattern, `\2` is the second remembered pattern, etc.
- `sed` has up to nine remembered patterns.
- Example, keep the first word of a line:

```
$ sed 's/\([a-z]*\).*/\1/'
```

- Regular expressions are greedy, and try to match as much as possible.
- `[a-z]*` matches zero or more lower case letters, and tries to match as many characters as possible.
- The `.*` matches zero or more characters after the first match.
- Since the first regex grabs all of the contiguous lower case letters, the second regex matches anything else:

Keep Part of the Pattern II

```
$ echo abcd123 | sed 's/\([a-z]*\).*/\1/'
abcd
```

- If you want to switch two words around, you can remember two patterns and change the order around:

```
$ sed 's/\([a-z]*\) \([a-z]*\)/\2 \1/'
```

- Note that there is a space between the two remembered patterns.
- This is used to make sure two words are found.
- However, this will do nothing if a single word is found, or any lines with no letters.
- You may want to insist that words have at least one letter by using:

```
$ sed 's/\([a-z][a-z]*\) \([a-z][a-z]*\)/\2 \1/'
```

Keep Part of the Pattern III

- or by using extended regular expressions (note that parentheses no longer need to have a backslash):

```
$ sed -r 's/([a-z]+) ([a-z]+)/\2 \1/'
```

- The `\1` does not have to be in the replacement string (in the right hand side).
- It can be in the pattern you are searching for (in the left hand side).
- If you want to keep only the first occurrence of two consecutive duplicated words:

```
echo "hello hello how are you?" | sed 's/\([a-z]*\) \1/\1/'
hello how are you?
```

Print Only Some Lines

- By default, `sed` prints every line.
- If it makes a substitution, the new text is printed instead of the old one.
- If you use `sed -n` it will not, by default, print any new lines.
- When the `-n` option is used, the `p` flag will cause the modified line to be printed.
- For example, if you want to print the lines with duplicated words:

```
$ sed -n '/\([a-z][a-z]*\) \1/p'
```

- Another example is:

```
$ sed -n 's/pattern/&/p' <file
```

- The above command is one way to duplicate the function of `grep` with `sed`.

Outline

Outline

If I

- The clause `if` is the most basic form of conditional.
- The syntax is:
  `if` **expression** `then` **statement1** `else` **statement2** `fi`.
- Where **statement1** is only executed if **expression** evaluates to true and **statement2** is only executed if **expression** evaluates to false.
- Examples:

```
if [ -e /etc/file.txt ]
  then
  echo "/etc/file.txt exists"
  else
  echo "/etc/file.txt does not exist"
fi
```

- In this case, the expression uses the option `-e file`, which evaluates to true only if “file” exists.
- Be careful, you must leave spaces between “[” and “]” and the expression inside.

If II

- With the symbol "!" you can do inverse logic.
- We can also create expressions that always evaluate to true or false:

```
if true
  then
  echo "this will always be printed"
  else
  echo "this will never be printed"
fi
if false
  then
  echo "this will never be printed"
  else
  echo "this will always be printed"
fi
```

- Other operators for expressions are the following.

If III

- **File Operators:**

```
[ -e filename ] true if filename exists
[ -d filename ] true if filename is a directory
[ -f filename ] true if filename is a regular file
[ -L filename ] true if filename is a symbolic link
[ -r filename ] true if filename is readable
[ -w filename ] true if filename is writable
[ -x filename ] true if filename is executable
[ filename1 -nt filename2 ] true if filename1 is more recent than filename2
[ filename1 -ot filename2 ] true if filename1 is older than filename2
```

- **String comparison:**

```
[ -z string ] true if string has zero length
[ -n string ] true if string has nonzero length
[ string1 = string2 ] true if string1 equals string2
[ string1 != string2 ] true if string1 does not equal string2
```

If IV

- **Arithmetic operators:**

```
[ X -eq Y ] true if X equals Y
[ X -ne Y ] true if X is equal to Y
[ X -lt Y ] true if X is less than Y
[ X -le Y ] true if X is less or equal than Y
[ X -gt Y ] true if X is greater than Y
[ X -ge Y ] true if X is greater or equal than Y
```

- The syntax ((...)) for arithmetic expansion can also be used in conditional expressions.
- The syntax ((...)) supports the following relational operators: $==, !=, >, <, >=, y <=$.
- Example:

```
if ((VAR == Y * 3 + X * 2))
then
   echo "The variable VAR  is equal to Y * 3 + X * 2"
fi
```

- We can also use conditional expressions with the OR or with the AND of two conditions:

```
[ -e filename -o -d filename ]
[ -e filename -a -d filename ]
```

If V

- **In general it is a good practice to quote variables** inside your conditional expressions:

```
if [ $VAR = 'foo bar oni' ]
then
echo "match"
else
echo "not match"
fi
```

- The previous expression might not behave as you expect.
- If the value of `VAR` is “foo”, we will see the output “not match”, but if the value of `VAR` is ”foo bar oni”, bash will report an error saying ”*too many arguments*”.
- The problem is that spaces present in the value of VAR confused bash.
- In this case, the correct comparison is:

```
if [ "$VAR" = 'foo bar oni' ]
```

- Recall that you have to use double quotes to use the value of a variable (i.e. to allow parameter expansion).

If with Regular Expressions

- We can use **glob patterns** and **regex** in conditional statements.
- Syntax `[[ ... ]]`
- When comparing with glob patters we have to use `==`
- When comparing with regex we have to use `=~`.
- Examples:

```
[[ $a == z* ]]   # True if $a starts with an "z" (pattern matching).
[[ $a == "z*" ]] # True if $a is equal to z* (literal matching).
```

- In the previous example, note that quotes deactivate the meaning of the glob pattern (also happens with regex).
- Another example with globbing:

```
if [[ $VAR == ??grid* ]] ; then echo $VAR; fi
```

- The same example using a regex:

```
if [[ $VAR =~ ^..grid.*$ ]] ; then echo $VAR; fi
```

Conditions on Execution I

- Each command has a return value or exit status.
- Exit status is used to check the result of the execution of the command.
- If the exit status is zero, this means that the command was successfully executed.
- The special variable $? is a shell built-in variable that contains the exit status of the last executed command.
- For a script, $? returns the exit status of the last executed command in the script or the number after the keyword `exit`:

```
command
if [ "$?" -ne 0]
 then
 echo "the previous command failed"
 exit 1
fi
```

- We can also replace the previous code by:

```
$ command || echo "the previous command failed"; exit 1
```

Conditions on Execution II

- We can also use the return code of conditions.
- Conditions have a return code of 0 is the condition is true or 1 if the condition is false.
- Using this, we can get rid of the keyword `if` in some conditionals:

```
$ [ -e filename ] && echo "filename exists" || echo "filename does not exist"
```

- Based on this return code, the echo is executed.

Outline

For I

- A `for` loop is classified as an iteration statement.
- The syntax is:

```
for VARIABLE in item1 item2 item3 item4 item5 ... itemK
do
        command1
        command2
        ...
        commandN
done
```

- The previous `for` loop executes K times a set of N commands.
- You can also use the values (item1, item2, etc.) that takes your control VARIABLE in the execution of the block of commands:

```
#!/bin/bash
for X in one two three four
do
echo number $X
done
```

For II

- for accepts any list after the key word "*in*":

```
#!/bin/bash
for FILE in /etc/r*
do
if [ -d "$FILE" ]
then
echo "$FILE (dir)"
else
echo "$FILE"
fi
done
```

- Furthermore, we can use *filename expansion* to create the list:

```
for FILE in /etc/r??? /var/lo* /home/student/* /tmp/${MYPATH}/*
do
cp $FILE /mnt/mydir
done
```

- We can use relatives paths too:

```
for FILE in ../* documents/*
do
echo $FILE is a silly file
done
```

For III

- In the previous example the expansion is relative to the script location.
- We can make loops with the positional parameters of the script as follows:

```
#!/bin/bash
for THING in "$@"
do
echo You have typed: ${THING}.
done
```

- With the command `seq` or with the syntax (()) we can generate C-styled loops:

```
#!/bin/bash
for i in `seq 1 10`;
do
echo $i
done
```

```
#!/bin/bash
for (( i=1; i < 10; i++));
do
echo $i
done
```

Outline

While

- A `while` executes a code block while a expression is true:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
        echo $X
        X=$((X+1))
done
```

- The loop is executed while the variable `X` is less or equal (-le) than 20.
- We can also create infinite loops, example:

```
#!/bin/bash
while true
 do
  sleep 5
  echo "Hello I waked up"
 done
```

Outline

Case I

- A `case` construction is a bash programming language statement which is used to test a variable against set of patterns.
- Often `case` statement let's you express a series of if-then-else statements that check single variable for various conditions or ranges in a more concise way.
- The generic syntax of `case` is the following:

```
case VARIABLE in
    pattern1)
        1st block of code ;;
    pattern2)
        2nd block of code ;;
    ...
esac
```

- The pattern is a glob pattern and it can actually be formed of several subpatterns separated by pipe character "|".
- If the VARIABLE matches one of the patterns (or subpatterns), its corresponding code block is executed.

Case II

- The patterns are checked in order until a match is found; if none is found, nothing happens.
- For example:

```
#!/bin/bash
for FILE in $*; do
  case $FILE in
   *.jpg | *.jpeg | *.JPG | *.JPEG)
   echo The file $FILE seems a JPG file.
   ;;
  *.avi | *.AVI)
   echo "The filename $FILE has an AVI extension"
   ;;
  -h)
   echo "Use as: $0 [list of filenames]"
   echo "Type $0 -h for help" ;;
  *)
   echo "Using the extension, I don't now which type of file is $FILE."
   echo "Use as: $0 [list of filenames]"
   echo "Type $0 -h for help" ;;
  esac
done
```

- The final pattern is *, which is a catchall for whatever didn't match the other cases.

Outline

printf I

- `printf` is a (more complete) command for generating outputs:

```
$ printf "hello printf"
hello printf$
```

- No new line had been printed out as it it in case of when using default setting of `echo` command.
- To print a new line:

```
$ printf "Hello, $USER.\n\n"
```

- or

```
$ printf "%s\n" "hello printf"
hello printf
```

- The format string is applied to each argument:

```
$ printf "%s\n" "hello printf" "in" "bash script"
hello printf
in
bash script
```

printf II

- %s is used as a format specifier to to print all argument in literal form.
- The specifiers are replaced by their corresponding arguments:

```
$ printf "%s\t%s\n" "1" "2 3" "4" "5"
1       2 3
4       5
```

- The %b specifier is essentially the same as %s but it allows us to interpret escape sequences with an argument:

```
$ printf "%s\n" "1" "2" "\n3"
1
2
\n3
$ printf "%b\n" "1" "2" "\n3"
1
2

3
$
```

printf III

- To print integers, we can use the %d specifier:

```
$ printf "%d\n" 255 0xff 0377 3.5
255
255
255
bash: printf: 3.5: invalid number
3
```

- %d specifiers refuses to print anything than integers.
- To `printf` floating point numbers use %f:

```
$ printf "%f\n" 255 0xff 0377 3.5
255.000000
255.000000
377.000000
3.500000
```

- The default behavior of %f `printf` specifier is to print floating point numbers with 6 decimal places.

printf IV

- To limit a decimal places to 1 we can specify a precision in a following manner:

```
$ printf "%.1f\n" 255 0xff 0377 3.5
255.0
255.0
377.0
3.5
```

- Formatting to three places with preceding with 0:

```
$ for i in $( seq 1 10 ); do printf "%03d\t" "$i"; done
001     002     003     004     005     006     007     008     009     010
```

- You can also print ASCII characters using their hex or octal notation:

```
$ printf "\x41\n"
A
$ printf "\101\n"
A
```

Outline

Outline

Functions I

- As with most programming languages, you can define functions in bash.
- The syntax is:

```
function function_name {
commands...
}
```

- or

```
function_name () {
commands...
}
```

- Functions can accept arguments in a similar way as the script receives arguments from the command-line:

```
#!/bin/bash
# file: myscript.sh
zip_contents() {
 echo "Contents of $1: "
  unzip -l $1
}
zip_contents $1
```

Functions II

- Note that we use the first positional parameter received in the command-line as the first argument for the function.
- The other special variables have also a meaning related to the function.
- E.g. $? returns the exit status of the last command executed in the function or the value after `return` or `exit`.
- The difference between `return` and `exit` is that the former does not finish the execution of the script, while the latter exits the script.
- A function may be compacted into a single line:

```
$ zip_contents () { echo "Contents of $1: "; unzip -l $1; }
```

- A semicolon must follow each command (also the final command).

Outline

Introduction

- Unlike many other programming languages, by default bash does not segregate its variables by type.
- Essentially, bash variables are character strings.
- Depending on context, bash permits arithmetic operations and comparisons on variables.
- The determining factor is whether the value of a variable contains only digits or not.
- You can explicitly define a variable as numerical using the bash built-in `declare`:

```
declare -i VAR
```

- We will use the convention of defining the name of the variables in **capital letters** to distinguish them easily from commands and functions.
- Variables have one of the following scopes: **shell** variables, **local** variables, **environment** variables, **position** variables and **special** variables.

Shell Variables

- A shell variable, as its name states, is a variable defined for a shell (e.g. a bash).
- In general, when you define a variable in a script, the variable is defined for the shell that is executing the script.
- To execute scripts, **the bash in which the script is launched clones itself** and the variables defined in the script are valid **only in this cloned shell**.
- You can also define a shell variable in the shell that is attached to a terminal:

```
$ VAR=hello
$ echo $VAR
```

- However, the previous variable **will not available to child programs** like a cloned bash that executes a script.
- If you need that a certain variable **is available for a child program**, you have to make it an **environment variable** as described later.

Local Variables I

- Local variables are those variables used **inside a function**.
- In a bash script, when you create a variable inside a function, it is added to the script's namespace.
- This means that if we set a variable inside a function with the same name as a variable defined outside the function, we will override the value of the variable.
- Furthermore, a variable defined inside a function will exist after the execution of the function.
- Let's illustrate this issue with an example:

```
#!/bin/bash
VAR="Outside the function"
my_function(){
VAR="Inside the function" }

my_function
echo $VAR
```

- When you run this script the output is "Inside the function".

Local Variables II

- If you really want to declare VAR as a local variable, whose scope is only its related function, you have to use the keyword `local`:

```
#!/bin/bash
VAR="Outside the function"

my_function(){
local VAR="Inside the function"
}

my_function
echo $VAR
```

- In this case, the output is ”Outside the function“.

Environment Variables I

- shell variables are not available for child processes like the bash that executes a script.
- Child processes only inherit the "**exported context**". Variables (and functions) are not exported by default.
- To export variables and functions and view the exported and current environment you can use the following commands:
  - **Export a variable**. Syntax: `export VAR` or `declare -x VAR`.
  - **Export a function**. Syntax: `export -f function_name`.
  - **View current context**. Syntax: `declare`.
  - **View exported context**. Syntax: `export` or `declare -x`.
- For example, let's use a script to test environment variables:

```
#!/bin/bash
echo $VAR
```

- Then, execute the following:

Environment Variables II

```
$ VAR=hello ; echo $VAR
hello
$ declare | grep VAR
VAR=hello
$ ./script.sh
$ export VAR ; export | grep VAR
MY_VAR=hello
$ ./script.sh
hello
```

- Notice that `script.sh` can use the variable '`VAR`' only after it is exported.
- As mentioned, exported variables and functions get passed on to child processes, but not-exported variables do not.
- Anyway, if VAR is defined inside the script, the value of the shell variable is the one used in that script.
- You can delete (unset) an environment variable (exported or not) using the command `unset`:

```
$ unset MY_VAR
$ ./env-script.sh
```

Initial Environment

- When a bash is executed, **a set of environment variables is loaded**.
- The two main files in which the system administrator or the user can set the initial environment variables for bash are:
  - ~/.bashrc (per user configuration).
  - /etc/profile (system-wide configuration).
- Typical environment variables widely used are:

Variable	Meaning
SHELL=/bin/bash	The shell that you are using.
HOME=/home/student	Your home directory.
LOGNAME=student	Your login name.
OSTYPE=linux-gnu	Your operating system.
PATH=/usr/bin:/sbin:/bin	Directories where bash will try to locate executables.
PWD=/home/student/documents	Your current directory.
USER=student	Your current username.
PPID=45678	Your parent PID.

`source` Command I

- As mentioned, when you execute a script, bash creates a child bash to execute the commands of the script.
- This is the default behavior because executing the scripts in this way, the parent bash is not affected by erroneously programmed scripts or by any other problem that might affect the execution of the script.
- In addition, the PID of the child bash can be used as the “PID of the script” to kill it, stop it, etc. without affecting the parent bash (which the user might have used to execute other scripts).
- While this default behavior is convenient most of the times, there are some situations in which is not appropriate.
- For this purpose, there exists a shell built-in called `source`.
- The `source` built-in allows executing a script without using any intermediate child bash.

`source` Command II

- `source` indicates that the current bash must directly execute the commands of the script:

```
#!/bin/bash
echo PID of our parent process $PPID
echo PID of our process $$
echo Showing the process tree:
pstree $PPID
```

- If you execute the script without `source`:

```
$ echo $$
2119
$ ./script.sh
PID of our parent process 2119
PID of our process 2225
Showing the process tree:
bash---bash---pstree
```

- Now, if you execute the script with `source`:

```
$ source ./script.sh
PID of our parent process 2114
PID of our process 2119
Showing the process tree from PID=2114:
gnome-terminal---bash---pstree
```

`source` Command III

- As you observe, when script.sh is executed with `source`, the bash does not create a child bash but executes the commands itself.
- An alternative syntax for `source` is a single dot.
- So the two following command-lines are equivalent:

```
$ source ./pids.sh
$ . ./pids.sh
```

- “Sourcing“ is a way of including variables and functions in a script from the file of another script.
- This is a way of creating an environment but without having to ”export“ every variable or function.
- We can create scripts that serve as a “library” of functions and variables.

Position Variables

- Position parameters are special variables that contain the arguments with which a script or a function is invoked.
- These parameters have already been introduced.
- A useful command to manipulate position parameters is `shift`:
  - It is a built-in and takes one argument, a number.
  - The positional parameters are shifted to the left by this number N.
  - The positional parameters from N+1 are renamed.

Special Variables

- Special variables indicate the status of a process.
- They are treated and modified directly by the shell so they are read-only variables.
- We have already used most of them.
- For example, the variable $$ contains the PID of the running process.

Variable	Meaning
$$	Process PID.
$*	String with all the parameters received.
$@	Same as above but treats each parameter as a different word.
$#	Number of parameters.
$?	Exit status (or return code) of last command (0=normal, >0=error).
$!	PID of the last process executed in background.
$_	Value of last argument of the command previously executed.