https://hpcleuven.github.io/Linux-scripting/

Linux scripting

Mag Selwa

SCAN ME

Overview

- I/O, pipes, redirections
- Scripts (bash)
- Variables and quotes
- Expressions
- Loops
- Arrays
- Some useful commands/features

- Hands-on: script out of control –> press **`Ctrl+C`** to terminate

Input and Output

- Programs and commands can contain an input and output. These are called ‘streams’. UNIX programming is oftentimes stream based.

- STDIN – ‘standard input,’ or input from the keyboard
- SDTOUT – ‘standard output,’ or output to the screen
- STDERR – ‘standard error,’ error output which is sent to the screen.

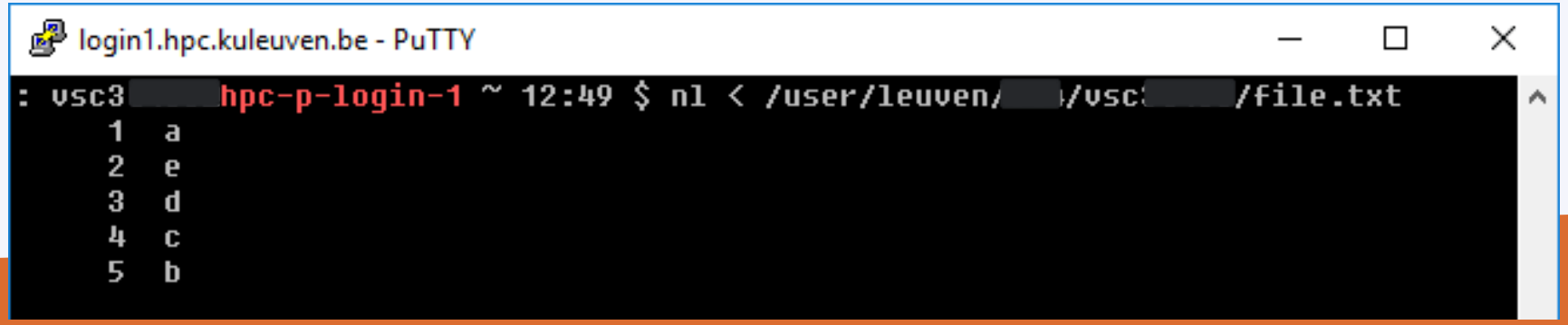
File Redirection

- Often we want to save output (stdout) from a program to a file. This can be done with the ‘redirection’ operator.
  - **myprogram** > *myfile* – using the ‘>’ operator we redirect the output from **myprogram** to file *myfile*

- Similarly, we can append the output to a file instead of rewriting it with a double ‘>>’
  - **myprogram** >> *myfile* – using the ‘>’ operator we append the output from **myprogram** to file *myfile*

Input Redirection

- Input can also be given to a command from a file instead of typing it to the screen, which would be impractical.
  - **mycommand** < *programinput* – using the '< ' operator we redirect the input from the file *programinput* to **mycommand**
  - *programinput* is printed to stdout, which is redirected to a command **mycommand.**
  - Not all commands read standard input (ls, date, who, pwd, cd, ps, …)



Redirecting stderr

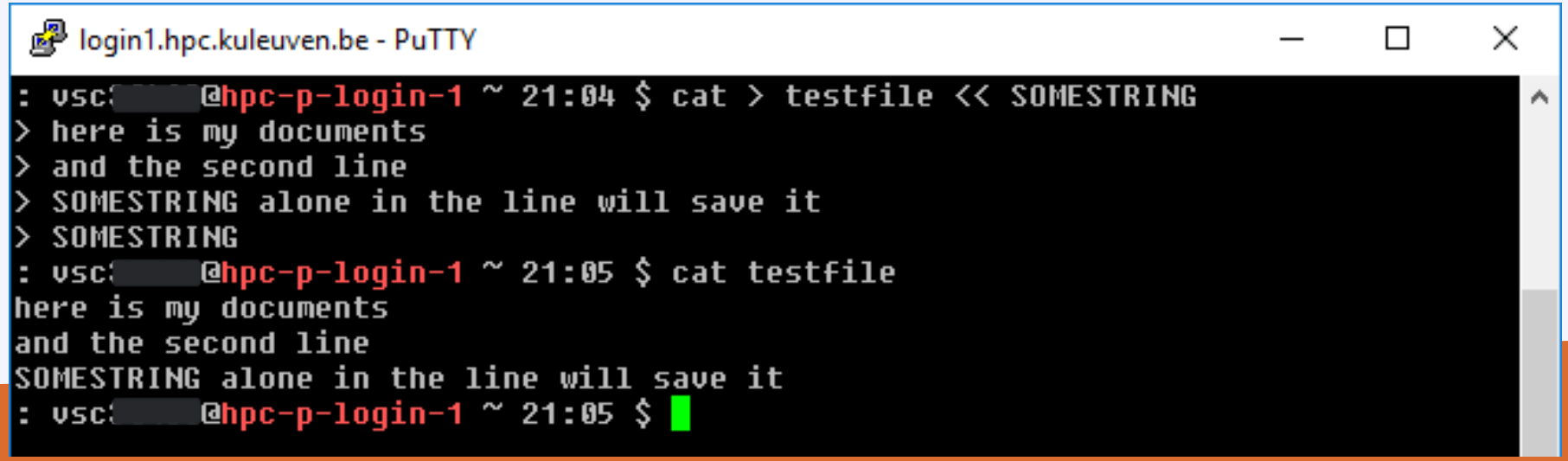
- Performing a normal redirection will not redirect sdterr. In Bash, this can be accomplished with ‘2>’
  - **command** *2> file1*

- Or, one can merge stderr to stdout (most popular) with ‘2>&1’
  - **command** *> file 2>&1*

Redirecting: here docs and here strings

- ‘Here docs’ are files created inline in the shell.
- The ‘trick’ is simple. Define a closing word, and the lines between that word and when it appears alone on a line become a file.
- Notice that:
  - the string could be included in the file if it was not ‘alone’ on the line
  - the string SOMEENDSTRING is more normally END, but that is just convention
- Lesser known is the ‘here string’:

```
$ cat > testfile <<< 'This file has one line'
```



Pipes

- Using a pipe operator '|' commands can be linked together. The pipe will link the standard output from one command to the standard input of another.

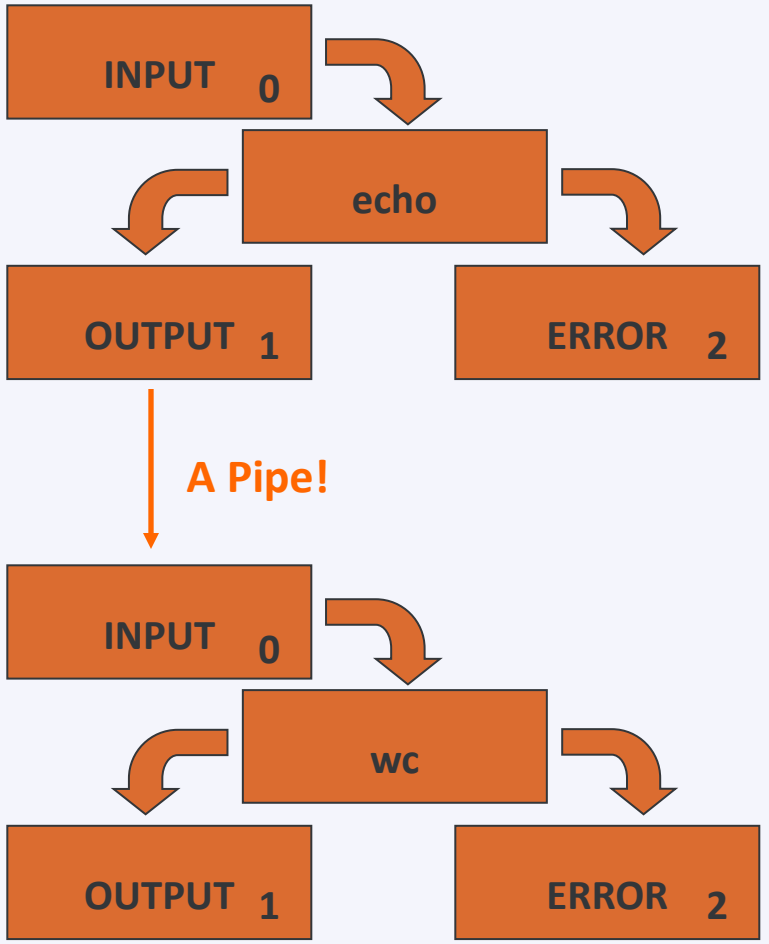
- Very helpful for searching files
- e.g. when we want to list the files, but only the ones that contain test in their name:

```
ls -la|grep test
```

Pipes

- Lots of Little Tools

```
echo “Hello” | \
  wc -c
```



Difference Between Single, Double, and Backwards Quote

- Single quotes (‘) do not interpret any variables
- Double quotes (“) interpret variables
- Backwards quotes (`) interpret variables and treat them as a program to run and return the results of that program

Quote characters

There are three different quote characters with different behaviour. These are:

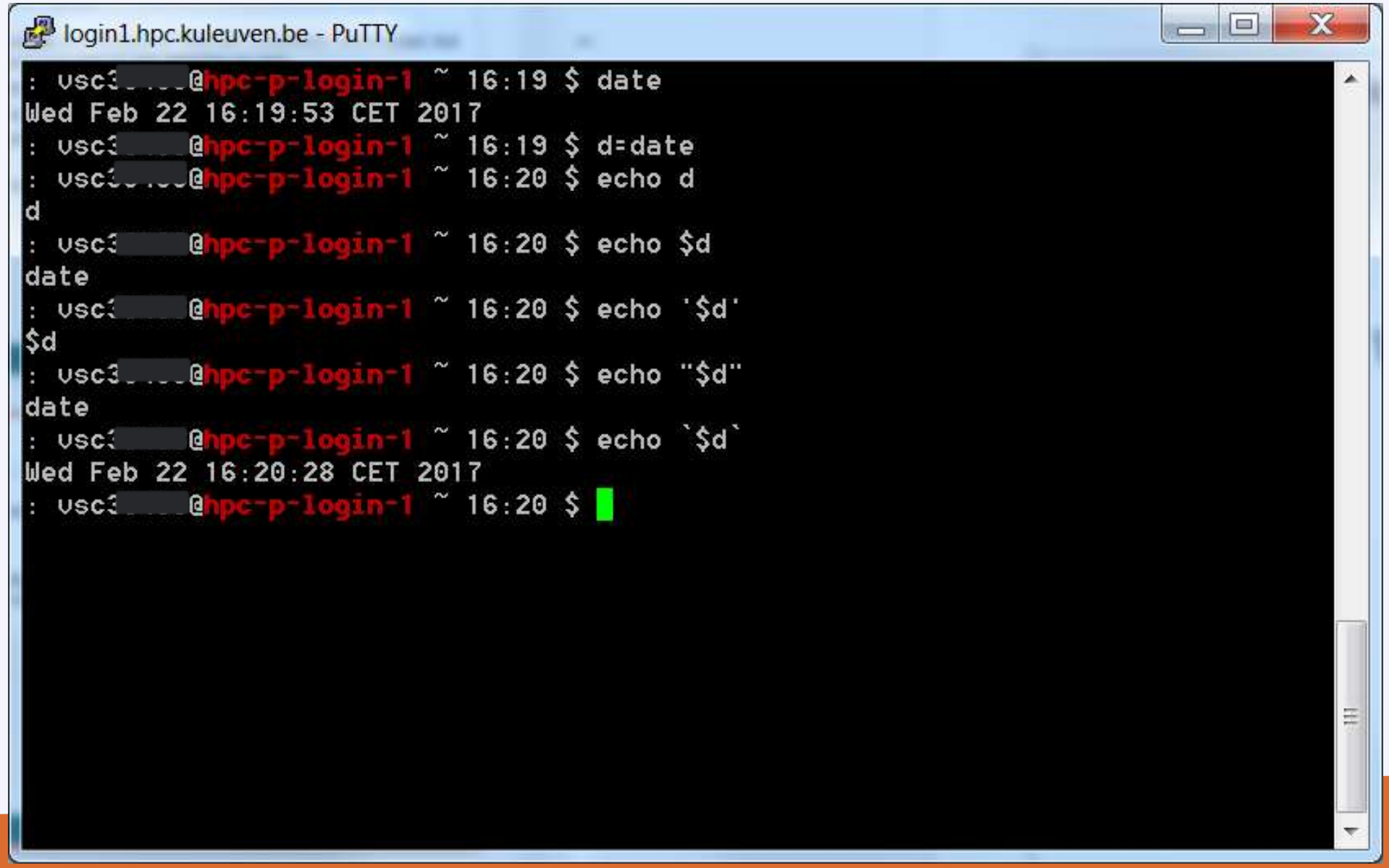
“ : double quote, weak quote. If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo “Today is:” `date``

Example Of Quote Difference



Printing/output

Basic use: `echo` (remember quotes!)

- `$ echo -n` -> do not output the trailing newline
- `$ echo -e` -> enable interpretation of backslash escapes
  - \n new line
  - \r carriage return
  - \t horizontal tab
  - \v vertical tab

```
: vsc[redacted]@tier2-p-login-3 ~ 15:49 $  echo -n "Enter your name:"
Enter your name:: vsc[redacted]@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\n"
Enter your name:

: vsc[redacted]@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\n and address:\n"
Enter your name:
 and address:

: vsc[redacted]@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\t and address:\n"
Enter your name:         and address:

: vsc[redacted]@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\v and address:\n"
Enter your name:
                and address:

: vsc[redacted]@tier2-p-login-3 ~ 15:50 $  echo -e "1\v2\v3\t4\t5"
1
 2
  3      4       5
: vsc[redacted]@tier2-p-login-3 ~ 15:50 $  echo -e "1\v2\v3\t4\r5"
1
 2
5 3      4
: vsc[redacted]@tier2-p-login-3 ~ 15:50 $
```

Printing/output

- Alternatively (when formatting needed): `printf`
- `printf [-v var] format [arguments]`

  Writes the formatted *arguments* to the standard output under the control of the *format*. The -v option causes the output to be assigned to the variable *var* rather than being printed to the standard output.
  - d - Format a value as a signed decimal number.
  - u - Format a value as an unsigned decimal number.
  - s - Format a value as a string.

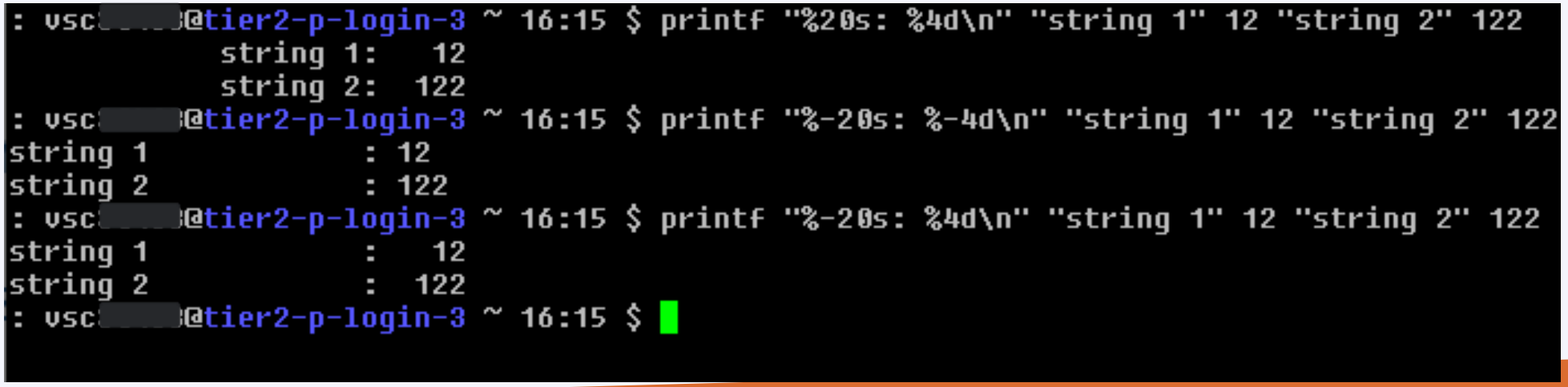
- Format specifiers can be preceded by a field width to specify the minimum number of characters to print. A **positive width** causes the value to be right-justified; a **negative width** causes the value to be left-justified. A width with a leading zero causes numeric fields to be zero-filled. Usually, you want to use negative widths for strings and positive widths for numbers.

- Precision: The precision for a floating- or double-number can be specified by using .<DIGITS>, where <DIGITS> is the number of digits for precision

- \n – new line

Printing/output

- `$ printf "%50s\n" "This field is 50 characters wide..."` -> will be printed as a string of 50 characters ended with the new line
- `$ printf "%20s: %4d\n" "string 1" 12 "string 2" 122` -> prints string 1 in 20 character fiels and corresponding number 12 where 4 digits are reserved for it. Next it starts new line and prints string 2 and number 122.
- Note that printf reuses the format if it runs out of format specifiers, which in the examples above allows you to print two lines (four values) with only two format specifiers.



Printing/output

- The default behaviour of `%f` specifier is to print floating point numbers with 6 decimal places. To limit a decimal places to 2 we can specify a precision in a following manner:
  `$ printf "%.2f\n" 255` -> will print: `255.00`

- Formatting to three places with preceding with 0, separated with tab:
  `$ printf "%03d\t" 2 3` -> will print: `002	003`

Printing/output

- How to create a table with multiple items?
- Definition of formats:

```
header="\n %-10s %8s %10s %11s\n"
format=" %-10s %08d %10s %11.2f\n"
```

- Values to print:

```
printf "$header" "ITEM NAME" "ITEM ID" "COLOR" "PRICE";printf
"$format"  Triangle 13 red 20 Oval 204449 "dark blue" 65.656 Square
3145 orange .7
```

```
: [redacted]@tier2-p-login-3 ~ 14:58 $ header="\n %-10s %8s %10s %11s\n"
: [redacted]@tier2-p-login-3 ~ 14:58 $ format=" %-10s %08d %10s %11.2f\n"
: [redacted]@tier2-p-login-3 ~ 14:58 $ printf "$header" "ITEM NAME" "ITEM ID" "COL
OR" "PRICE";printf "$format"  Triangle 13 red 20 Oval 204449 "dark blue" 65.656
Square 3145 orange .7

 ITEM NAME    ITEM ID       COLOR       PRICE
 Triangle    00000013         red       20.00
 Oval        00204449   dark blue       65.66
 Square      00003145      orange        0.70
```

Sequences

- `seq` - to print sequence of numbers
  - `seq [OPTION]... FIRST INCREMENT LAST`
- `-f, --format=FORMAT` (use printf style floating-point FORMAT)
- `-s, --separator=STRING` (use STRING to separate numbers, default: \n)
- `-w`, --equal-width

- `$ seq 1 10` -> prints sequence of numbers between 1 and 10 in new lines
- `$ seq 1 2 10` -> prints every other number between 1 and 10
- `$ seq -s "," -w 10 -1 1` -> prints decreasing numbers from 10 to 1, separated with “,” and each number in 2 digits decimal representation
- `$ seq -s " " -f "%4.2f" 1 10` -> prints sequence of numbers between 1 and 10, separated with space, with 2 digits precision

Sequences

```
: [redacted]@tier2-p-login-3 ~ 15:19 $ seq 1 10
1
2
3
4
5
6
7
8
9
10
: [redacted]@tier2-p-login-3 ~ 15:19 $ seq 1 2 10
1
3
5
7
9
: [redacted]@tier2-p-login-3 ~ 15:19 $ seq -s "," -w 10 -1 1
10,09,08,07,06,05,04,03,02,01
: [redacted]@tier2-p-login-3 ~ 15:19 $ seq -s " " -f "%4.2f" 1 10
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
: [redacted]@tier2-p-login-3 ~ 15:19 $
```

Reverse string

- `rev` - reverse lines of a file or files or a string

```
:          @tier2-p-login-3 ~ 15:24 $ var1=abcdefghijk
:          @tier2-p-login-3 ~ 15:24 $ echo $var1
abcdefghijk
:          @tier2-p-login-3 ~ 15:24 $ echo $var1|rev
kjihgfedcba
:          @tier2-p-login-3 ~ 15:24 $
```

- `tac` – print files in reverse

```
:          @tier2-p-login-3 ~ 15:26 $ more testfile
This is my test file
Line 1 AAA
Line 2 BBB
Line 3 CCC
Line 4 DDD
Line 5 EEE
:          @tier2-p-login-3 ~ 15:26 $ tac testfile
Line 5 EEE
Line 4 DDD
Line 3 CCC
Line 2 BBB
Line 1 AAA
This is my test file
:          @tier2-p-login-3 ~ 15:26 $
```

Bash scripts

To Script or not to Script

- Pros
  - File processing
  - Glue together compelling, customized testing utilities
  - Create powerful, tailor-made manufacturing tools
  - Cross-platform support
  - Custom testing and debugging

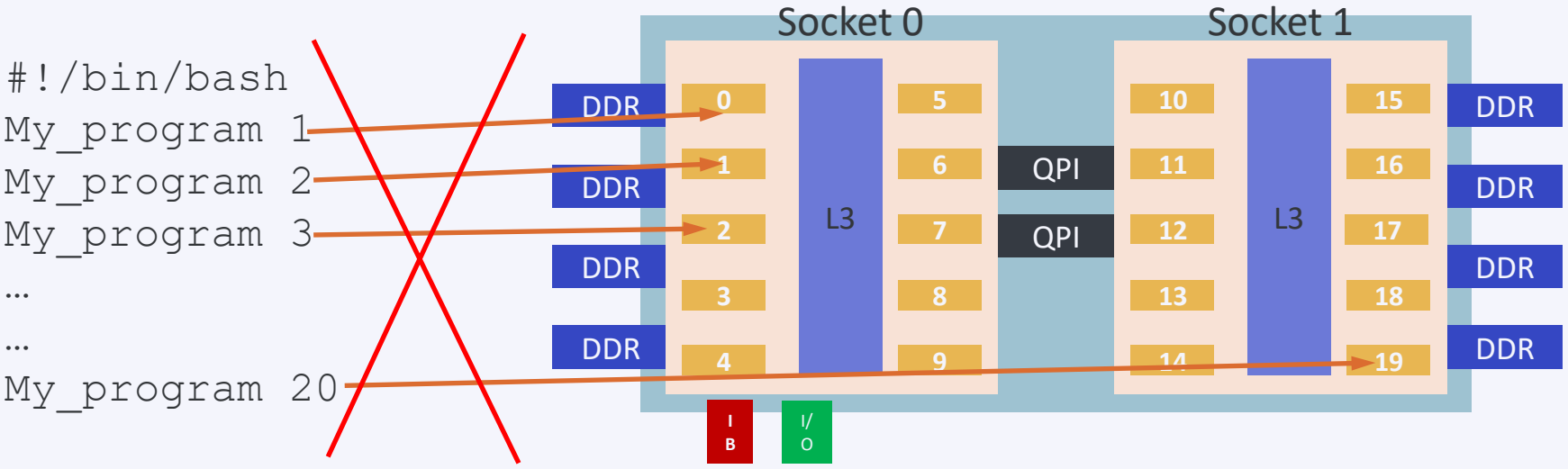
- Cons
  - Performance slowdown
  - Accurate scientific computing

So Many Commands…

- Unix has a lot of commands, but there is no way it has everything
- What do you do if no command exists that does what you want?
  - Build it yourself!
  - The shell itself is actually a programming language
  - A shell program consists of a sequential list of commands
  - Think of it as a listing of programs to run in order (interpreted language)

Sequential…

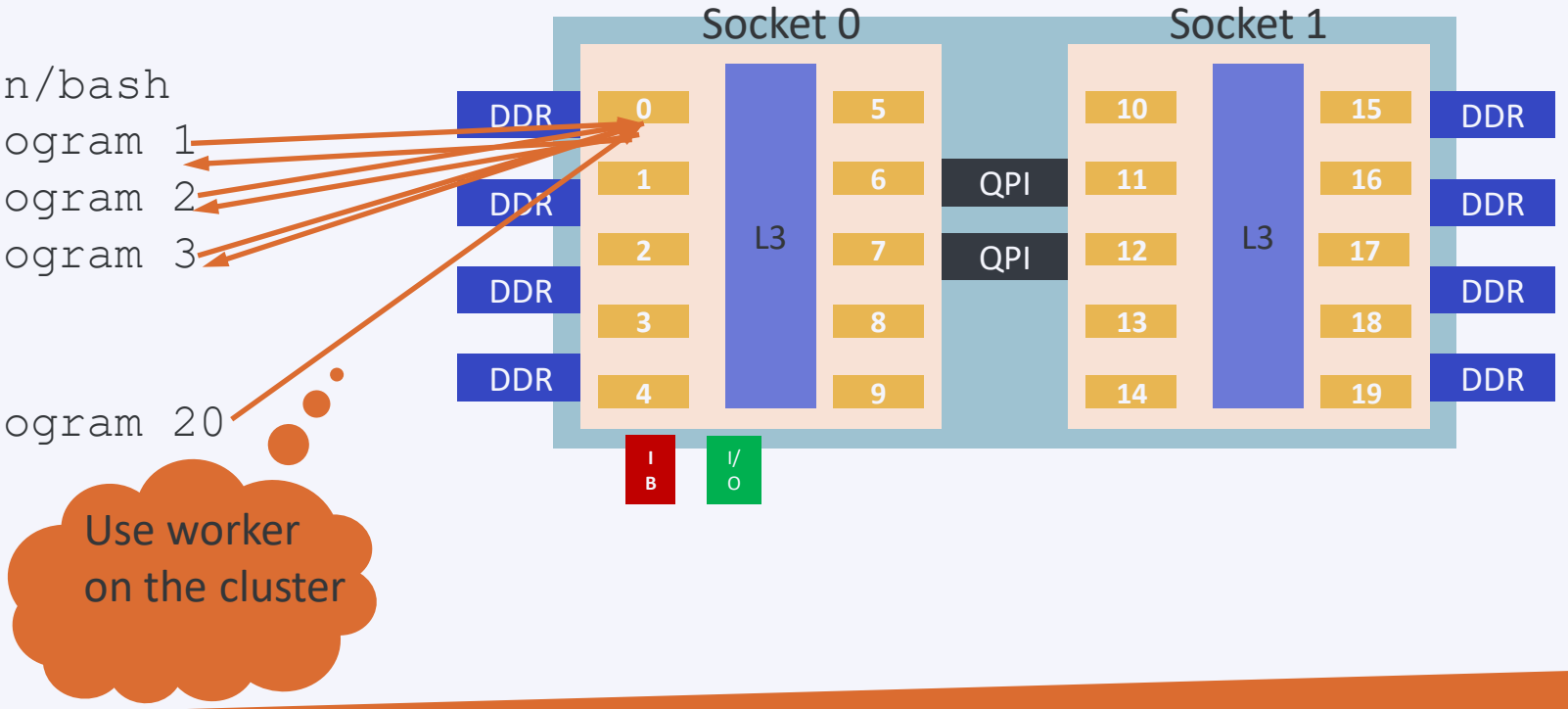
- A shell program consists of a sequential list of commands
- -> bash script on HPC cluster will run sequentially



Sequential…

- A shell program consists of a sequential list of commands
- -> bash script on HPC cluster will run sequentially

```
#!/bin/bash
My_program 1
My_program 2
My_program 3
…
…
My_program 20
```



When Not to Use Bash?

- Resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)
- Procedures involving heavy-duty math operations, especially floating point arithmetic arbitrary precision calculations, or complex numbers
- Cross-platform portability required
- Complex applications, where structured programming is a necessity (need type-checking of variables, function prototypes, etc.)
- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)
- Need native support for multi-dimensional arrays or data structures, such as linked lists or trees
- Need to generate or manipulate graphics or GUIs
- Need direct access to system hardware or port or socket I/O

First Thing: Comments

- A comment line in a shell script starts with the # symbol
- Example:
  - `# This is a comment`
- Comments are not processed and are only there to help people read your program
- Use them to explain your code
- A comment may be in the middle of a line, e.g.
  `Name='John Smith' # FirstName LastName`

Using the Shell

In a file

We can execute those commands in the order in which they appear in the file:

```
$ bash newscript
$ source newscript
$ . newscript
```

Run in a sub-shell (as a child process)

Run in the current process/shell/context.
Note: you need a shebang on top of the script

Using the Shell

As a shell script

BUT we can take the last step and start to create self-contained scripts that run on their own.

We will need to do two things:

1. Set Shebang to specify the CLI to use, and
2. Make our file executable

```
chmod u+x myscript.sh
```

Change mode

User (u) is granted execution (x) rights

The file extension (.sh) is meaningless (only reminder to us)

The “Shebang”

To specify that a file is to become a shell script you specify the interpreter like this at the very start of the file:

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

(or #!/usr/bin/perl or …)

This is known as the “Shebang” (#!).

Example

Now let's create a very simple shell script. This will simply echo back what you enter on the command line:

```
#!/bin/bash
echo Hello
```

Enter this in a file new.sh, then do:

```
$ chmod 755 new.sh
```

or

```
$ chmod u+x new.sh
```

To run the script do:

```
$ ./new.sh
```

chmod: changing permissions

- Permissions allow you to share files or directories or to lock them down to be private.
- `$ chmod` (change mode)
- `$ chmod <permissions> <files>`
  2 formats for permissions:
  - octal format (3 digit octal form)
  - symbolic format

chmod: changing permissions

- octal format (abc):
  a,b,c = r*4+w*2+x*1 (r, w, x: booleans)
  - 0 none ---
  - 1 execute-only --x
  - 2 write -w-
  - 3 execute and write -wx
  - 4 read-only r–
  - 5 read and execute r-x
  - 6 read and write rw-
  - 7 read, write, and execute rwx

- `$ chmod 644 <file>`
  (rw for u, r for g and o)

660 : 110 110 000
⇒ rw- rw- ---
540 : 101 100 000
⇒ r-x r-- ---

chmod: changing permissions

- symbolic format:
  `$ chmod go+r`: add read permissions to group and others.
  `$ chmod u-w`: remove write permissions from user.
  `$ chmod a-x`: (a: all) remove execute permission from all.

- `$ chmod u+rwx g+r`: add all permissions for the user, read permissions to group and none to the others.
  `$ chmod 740`

umask

- The user file creation mode mask (`umask`) is used to determine the file permission for newly created files. It can be used to control the default permissions of newly created files.

- Default umask is 022
  - Final permission for the files:

    666-022=644 (rw- r--r--)
  - Final permission for directories:

    777-022=755 (drwxr-xr-x)

Shell scripts

- Shell scripts are “programs” that are completely uncompiled, but read and executed by the shell line by line.
- Typically end in .sh (optional)
- Must be chmod’ed executable.
- Start with a “shebang” – tells the shell what to use to interpret it. e.g.,
  - `#! /bin/bash` for a bash script.

  Otherwise it can be called as

  ```
  $ bash newscript
  ```

Easy hello world program:

```
#! /bin/bash
echo “Hello World”
```

Return Values And exit

- Every program and shell returns an integer value (exit status) to indicate success or failure (mainly of the last command)
- Used to inform other programs how things went
- 0 = Success
- Any other number = Failure (different type)

Key concepts

- Upon exiting, every command returns an integer to its parent called a return value.
- The shell variable $? expands to the return value of previously executed command (e.g. 0 when success).

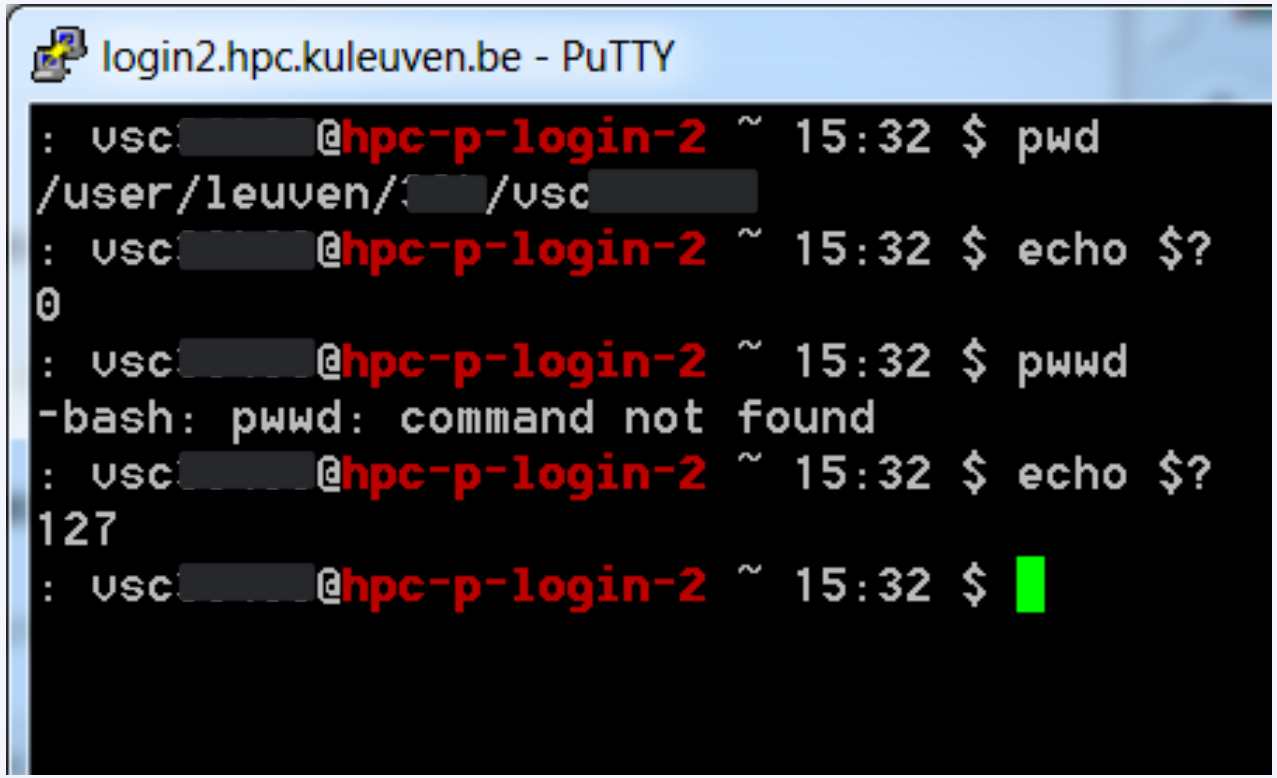
  compare:

  ```
  $ pwd
  $ echo $?
  ```

  with

  ```
  $ pwwd
  ```
  (does not exist, mistyped)

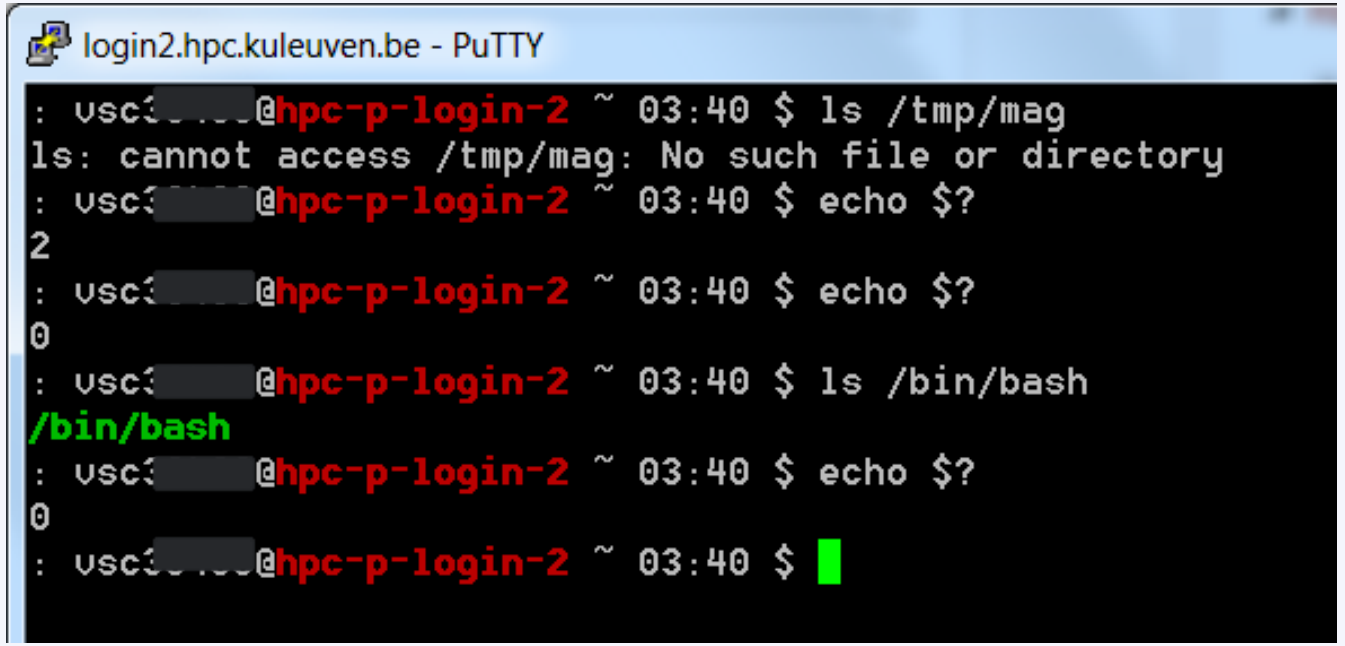
  ```
  $ echo $?
  ```



Exit Status

- `$?`
- `0` is True

```
$ ls /does/not/exist
$ echo $?
1
$ echo $?
0
```



Reserved exit codes

Exit Code Number	Meaning	Example	Comments
1	Catchall for general errors	let "var1 = 1/0"	Miscellaneous errors, such as "divide by zero" and other impermissible operations
2	Misuse of shell builtins (according to Bash documentation)	empty_function() {}	Missing keyword or command, or permission problem (and *diff* return code on a failed binary file comparison).
126	Command invoked cannot execute	/dev/null	Permission problem or command is not an executable
127	"command not found"	illegal_command	Possible problem with $PATH or a typo
128	Invalid argument to exit	exit 3.14159	**exit** takes only integer args in the range 0 - 255 (see first footnote)
128+n	Fatal error signal "n"	*kill -9* $PPID of script	**$?** returns 137 (128 + 9)
130	Script terminated by Control-C	*Ctl-C*	Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255*	Exit status out of range	exit -1	**exit** takes only integer args in the range 0 – 255. An exit value greater than 255 returns an exit code modulo 256. For example, *exit 3809* gives an exit code of 225 (3809 % 256 = 225).

Variables

- In order to hold values, variables are supported
- Any combination of letters, numbers, and _
  - Must start with a letter or _
- Case sensitive
  - `D` and `d` are two separate variables
- Different from filenames concept!

Setting And Using Variables

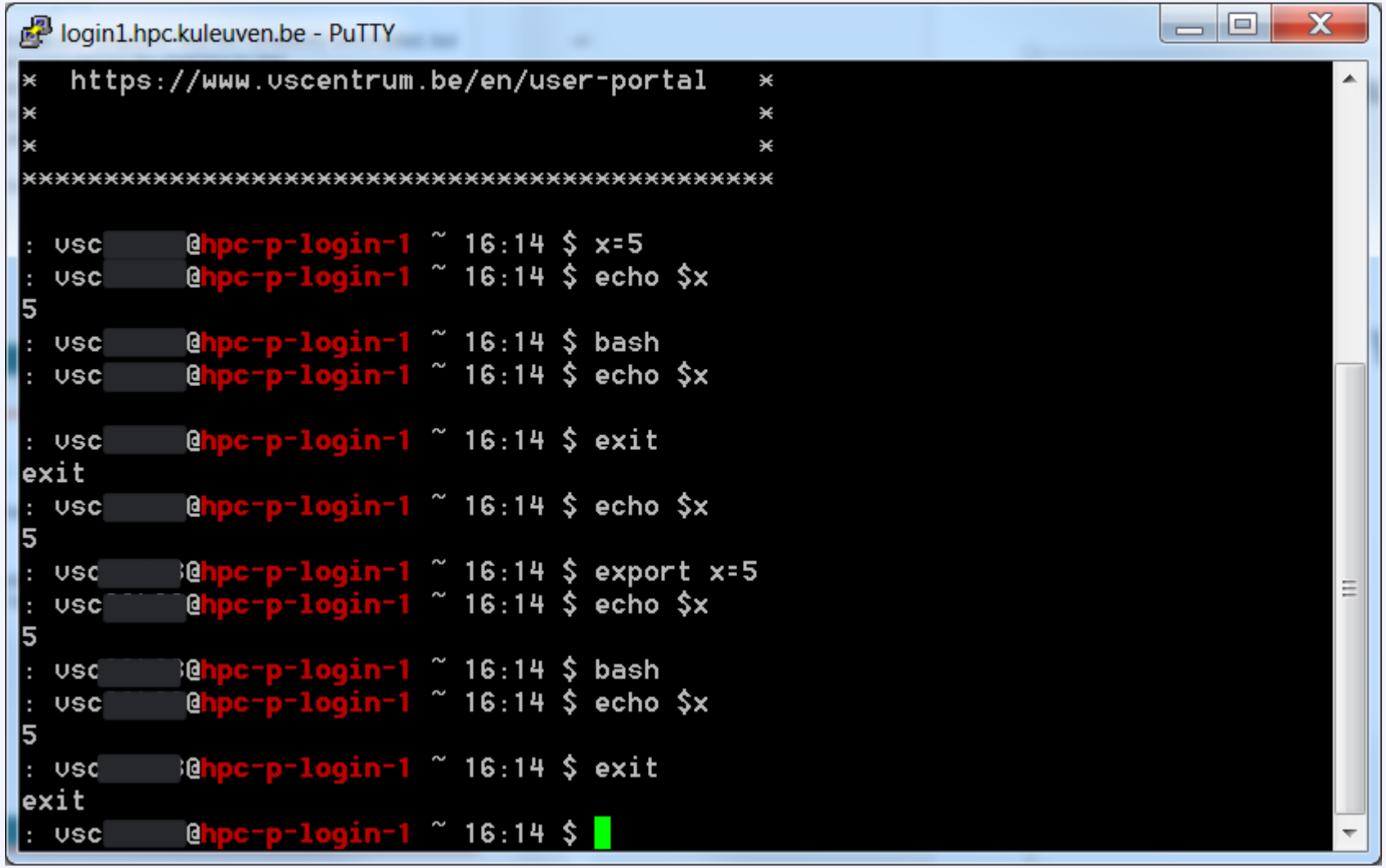
- Setting variables is done through the = operation
  - No prior declaration is needed
  - All variables are just strings of characters
  - `$ d=date`
  - `export` exports to children of the current process, by default they are not exported
  - `$ export d=date`
- Reading variables and accessing variables is done with the `$`
  - Variable is replaced with its value
  - `$ echo $d`
  - `$ echo ${d}`

Child process

- Each process may create many child processes but will have at most one parent process; if a process does not have a parent this usually indicates that it was created directly by the kernel.
- When a child process terminates, some information is returned to the parent process.
- When a child process terminates before the parent has called wait, the kernel retains some information about the process, such as its exit status, to enable its parent to call *wait* later. Because the child is still consuming system resources but not executing it is known as a zombie process.
- Structure of the processes can be checked with `pstree` command that shows running processes as a tree:

```
$ pstree vsc3XXXX
screen---bash---ssh
screen---bash---sudo---su---bash
sshd---bash---pstree
sshd---bash---ssh
sshd---bash---top
sshd---bash---gedit
```

Setting And Using Variables



Defining local variables

- As in any other programming language, variables can be defined and used in shell scripts.
- Unlike other programming languages, variables in Shell Scripts are not typed.
- Examples :

```
a=1234  # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1`  will perform arithmetic so b is 1235 now.
      Note : +,-,/,*,**,  %  operators are available.
b=abcde # b is string
b='abcde' # same as above but much safer.
b=abc  def # will not work unless 'quoted'
b='abc def'  # i.e. this will work.
```

IMPORTANT: DO NOT LEAVE SPACES AROUND THE =

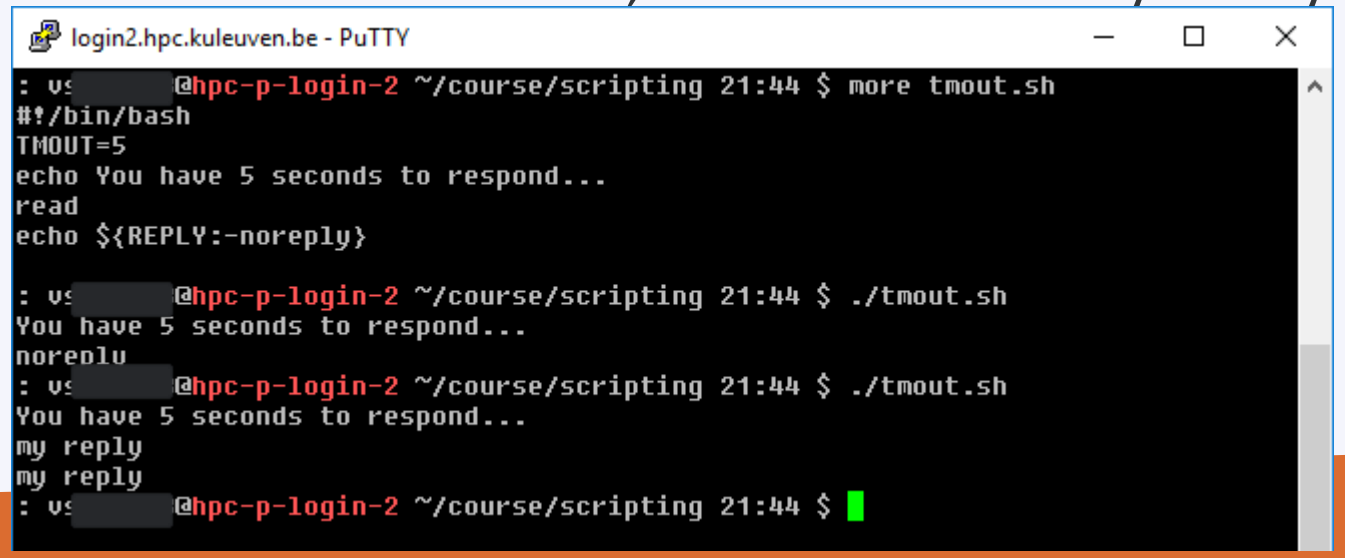
Reading User Input Into Variables

- REPLY

  No need to give a variable name for read

- TMOUT

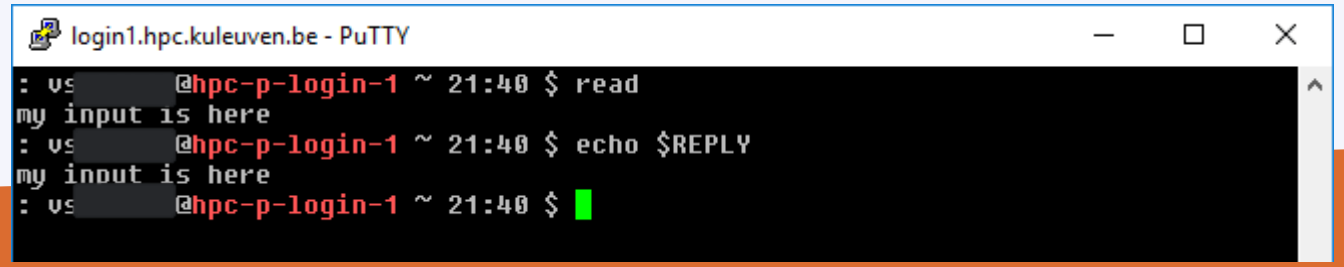
  You can timeout reads, which can be really handy



Reading User Input Into Variables

- Usage: `read var1 var2 …`
- Reads values from standard input and assigns them to each variable
- If more words are typed in then the excess gets assigned to the last variable
- If more variables are assigned than values given, the excess variables are empty
- REPLY

  No need to give a variable name for read



Example Of User Input

Command Line Parameters

- Just like all other Unix programs, shell scripts can read parameters of the command line
- Different from user input
  - Value is known before execution, not typed in during execution
- Example:
  - `$ testScript.sh testFile`

Special Variables For Command Line Parameters

- Accessing command line parameters requires special variables
  - `$0`

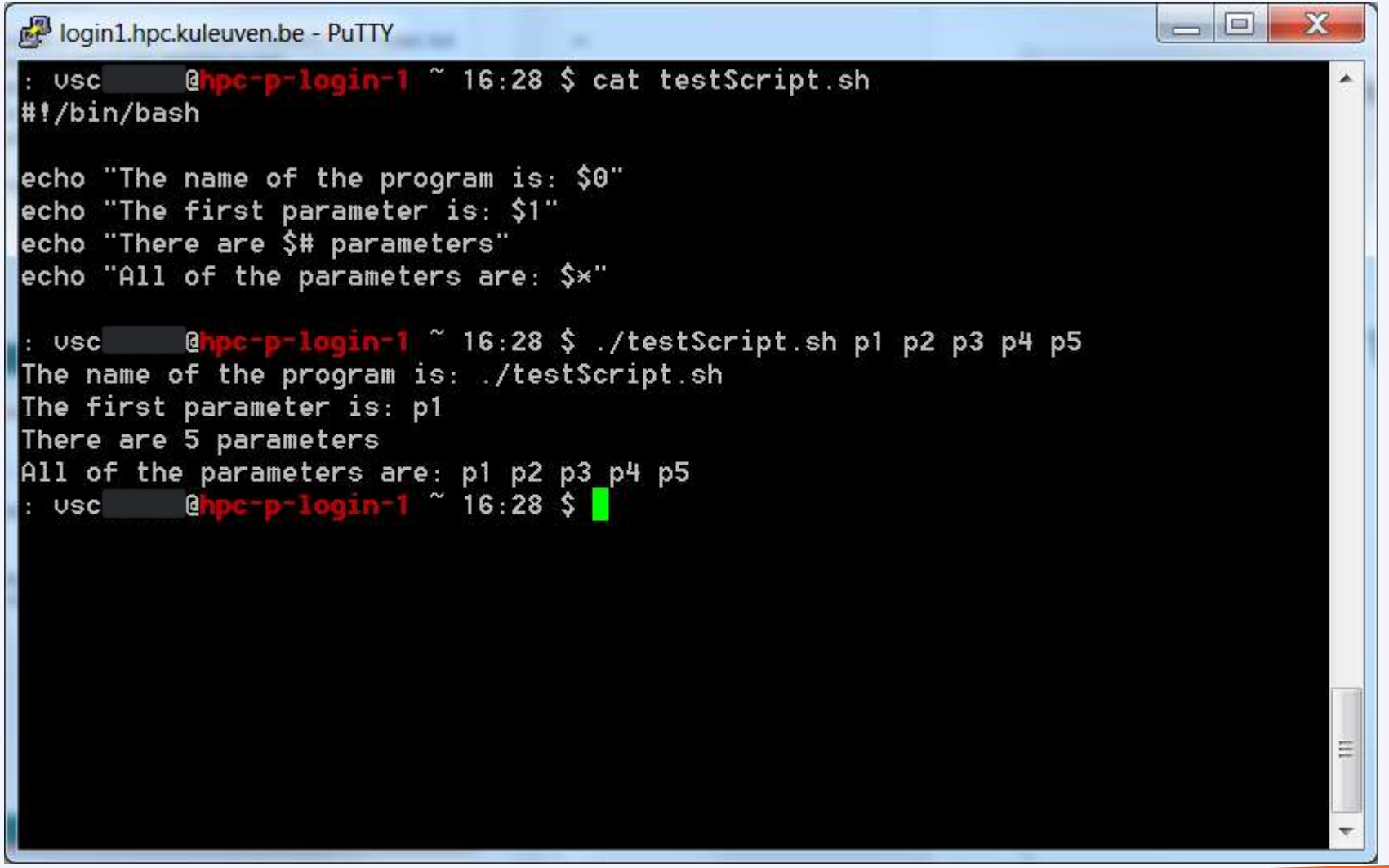
    The name of the running program
  - `$1-$9`

    The first nine arguments to the program
  - `$*` or `$@`

    All of the command line arguments
  - `$#`

    The total number of command line arguments

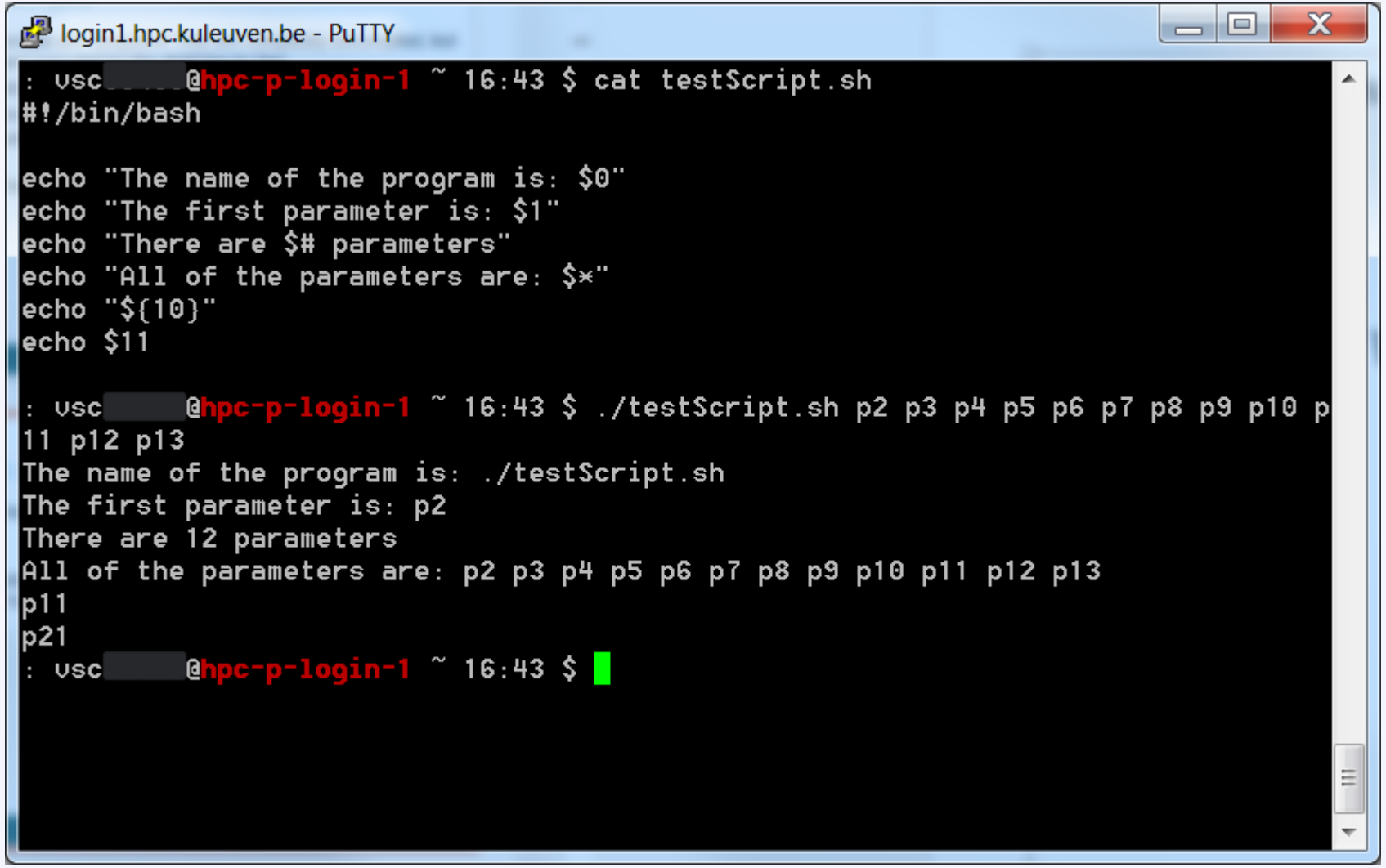
Example Of Command Line Parameters



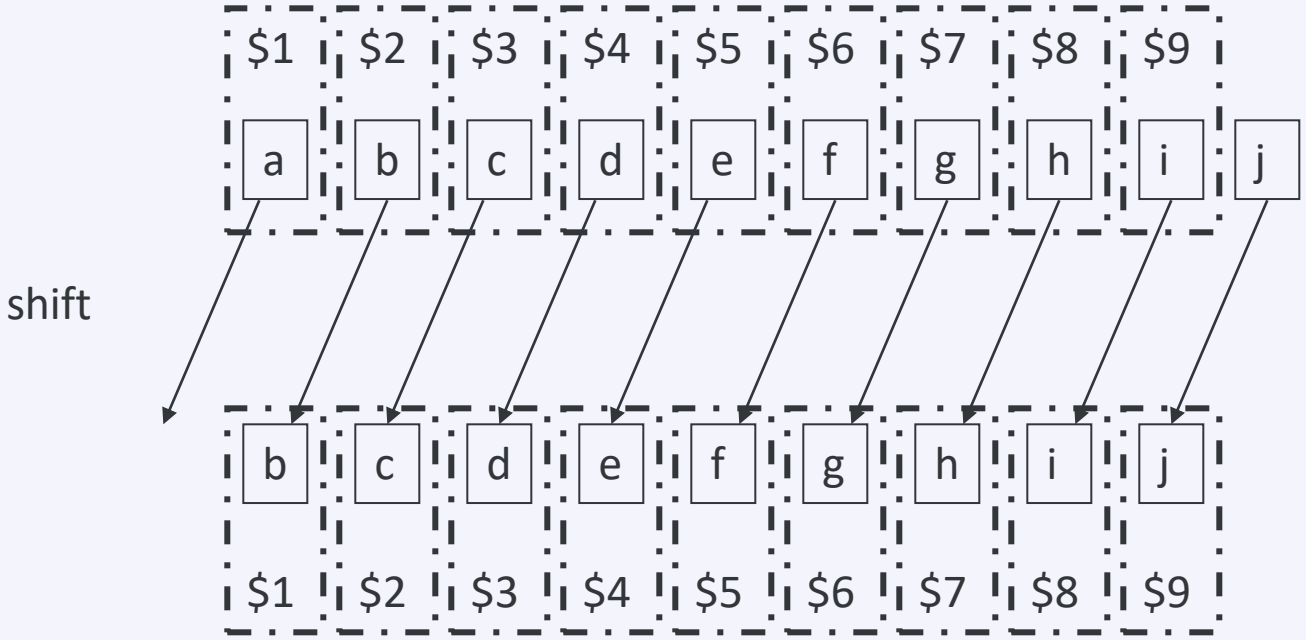
What If There Are More?

- Shell programs aren’t limited to only 9 arguments
- Unfortunately, you can only access 9 at a time
- Using `$var` instead of `${var}` is shorthand in bash. Bash internally treats variables that start with a digit as a "positional parameter." When bash detects a positional parameter it only looks at the first digit, e.g. `$10` returns $1"0". By calling `${10}` you are instructing bash to look at the complete variable instead of its built-in default of the first digit.
- `shift` command
  - Shifts the arguments to the left and brings new values in from the right

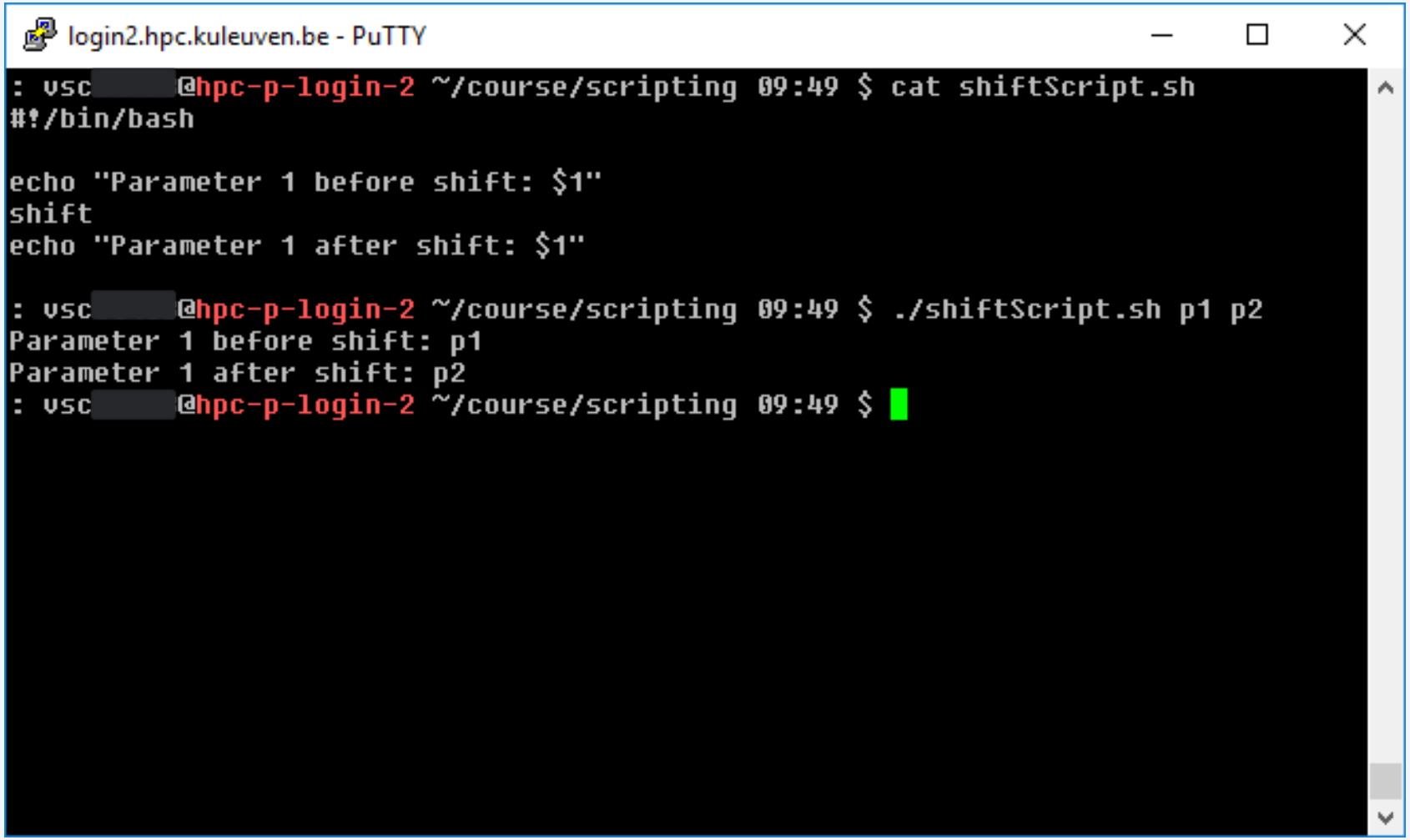
What If There Are More?



Visual Representation Of shift



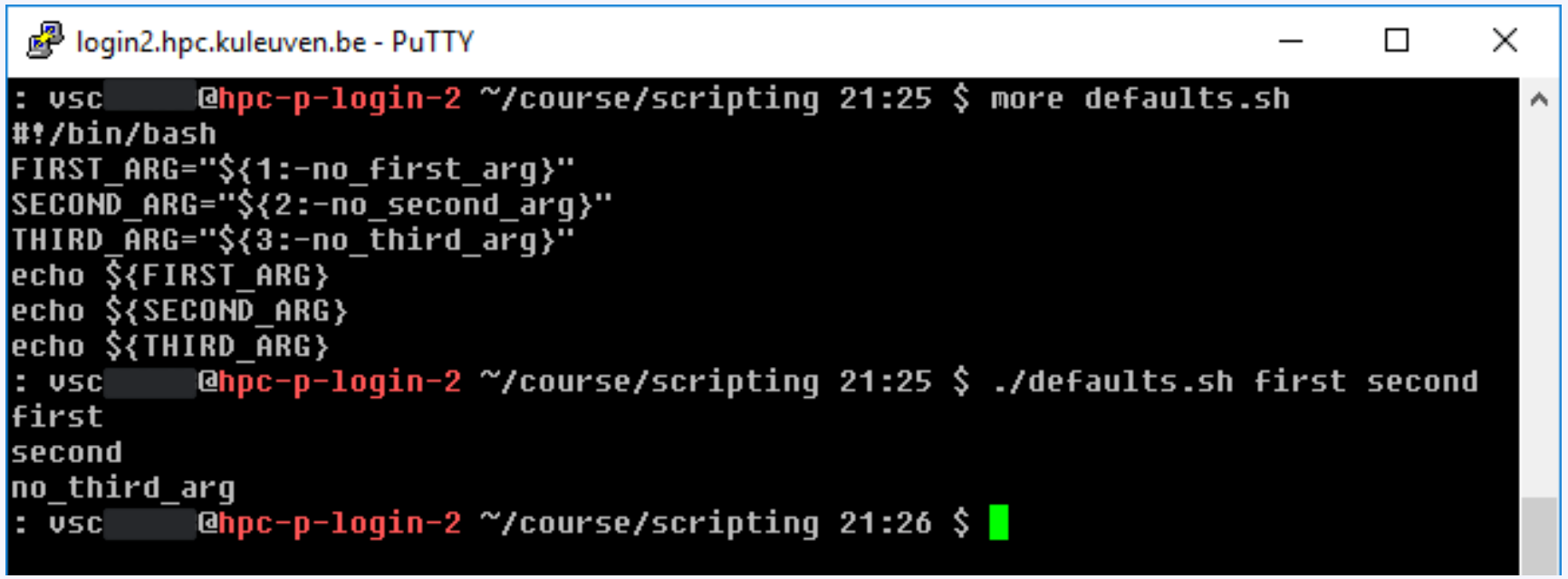
Example Of Shift



Command Line Parameters - defaults

If you have a variable that's not set, you can 'default' them by

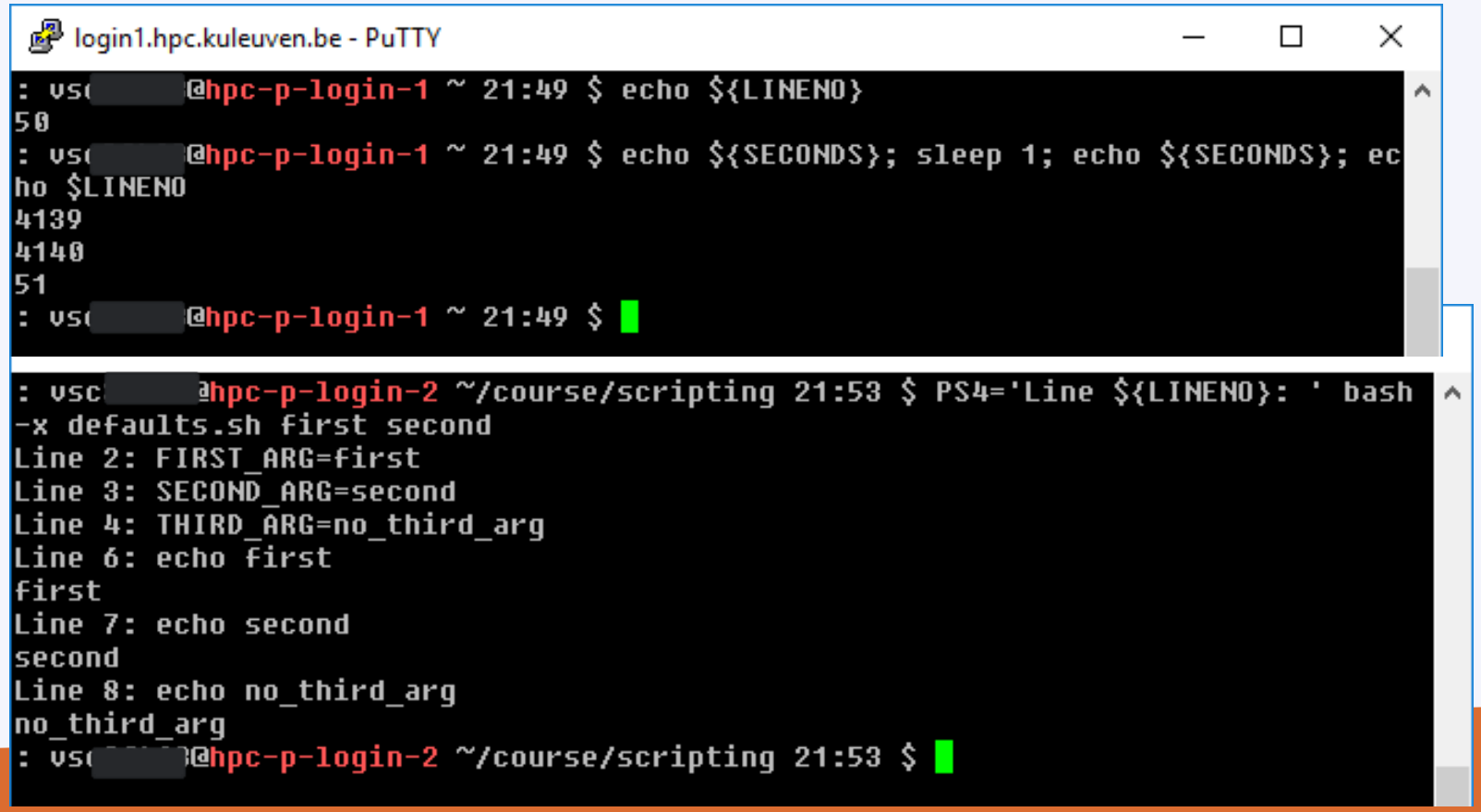
```
ARGUMENT="${1:-default_argument}"
```



Handy for debugging: LINENO and SECONDS

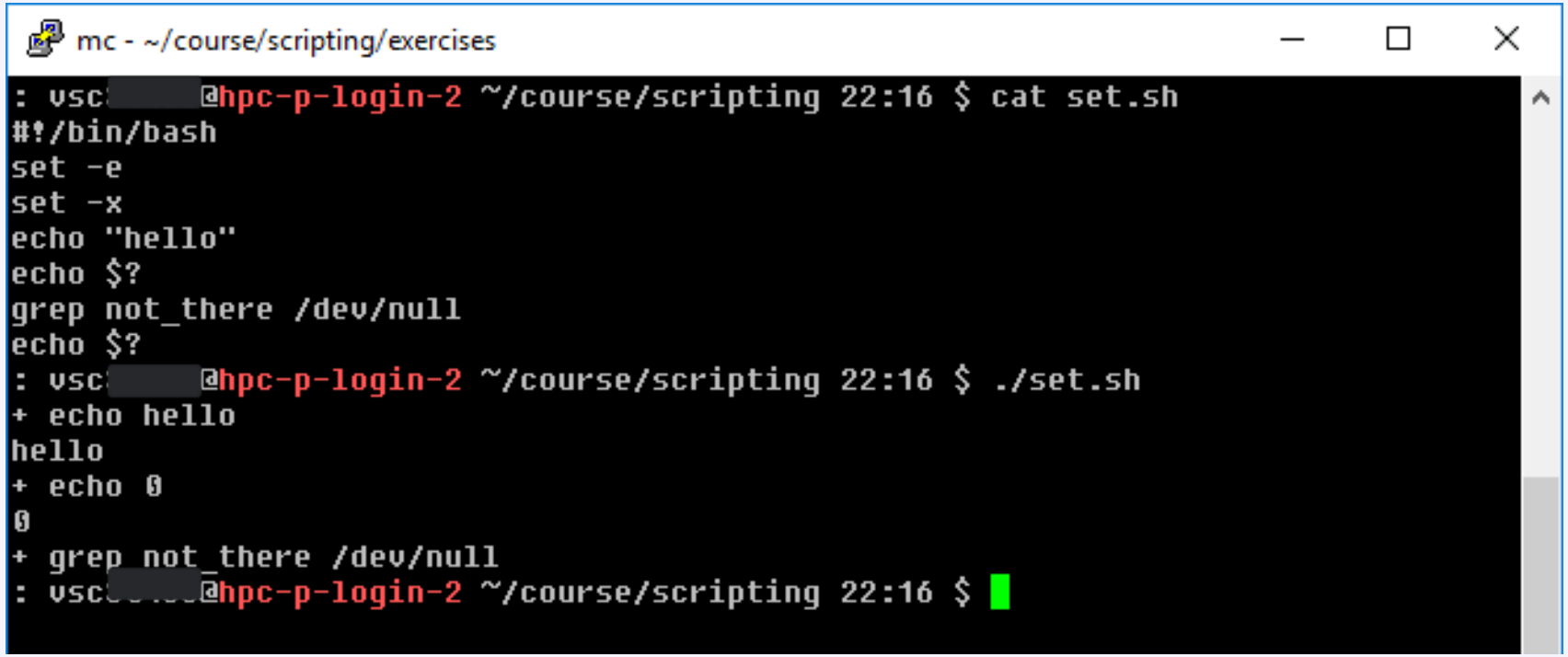
- The variable PS4 denotes the value of the prompt printed before the command line is echoed when the -x option is set and defaults to : followed by space.
- PS4 can be changed to emit the LINENO (The line number in the script or shell function currently executing).

```
$ PS4='Line ${LINENO}: ' bash -x script
```



Handy for debugging: set

- Bash has configurable options which can be set on the fly.
- `set -e`
  exits from a script if any command returned a non-zero exit code
- `set -x`
  outputs the commands that get run as they run



Hands-on 1

1. Write the simplest script that greets you with your name (e.g. *“Hello Mag!”*) after the execution – echo command and run it

2. Change your name to be taken from environment variables (`$USER`) and execute it

3. Check the exit code

4. Create a variable `today` that refers to the command `date`. Add to the first script a line that says *“Today is …..”* where the date is taken from the `today` variable.

5. Add an extra line to the first script that repeats given parameter in the sentence: *“Your input is: “*

Arithmetic and expr

- Variables that contain numbers can be treated as numbers
- You can perform the following operations on them:
  `+, -, \*, /, **, %`
- Shell programming is not good at numerical computation, it is good at text processing.
- However, the `expr` command allows simple integer calculations, e.g.

```
$ i=1
$ expr $i + 1
2
```

- To assign the result of an `expr` command to another shell variable, surround it with backquotes:

```
$ i=1
$ i=`expr $i + 1`
$ echo "$i"
2
```

expr

- The * character normally means “all the files in the current directory”, so you need a “\” to use it for multiplication:

```
$ i=2
$ i=`expr $i \* 3`
$ echo $i
6
```

- expr also allows you to group expressions, but the “(“ and “)” characters also need to be preceded by backslashes:

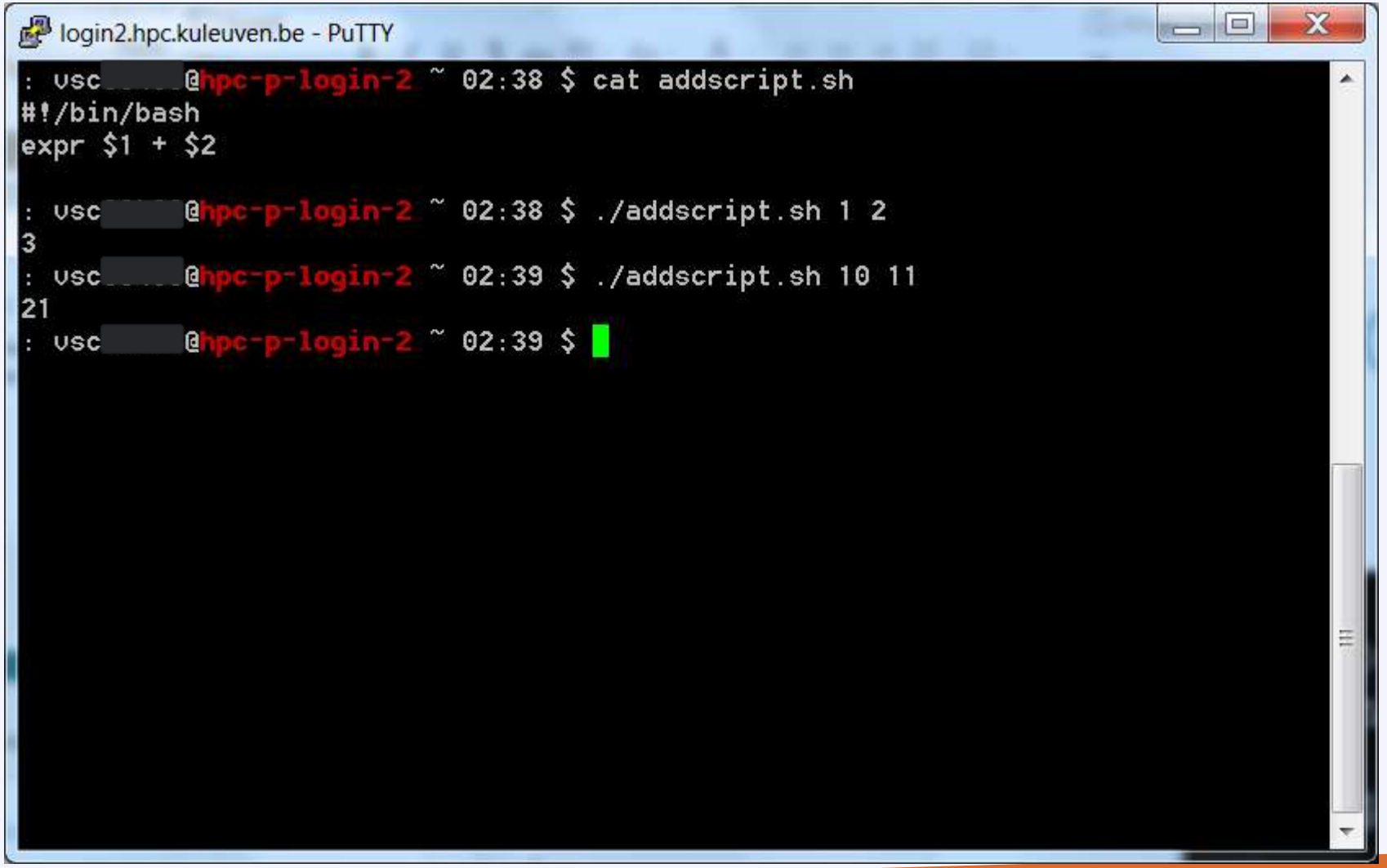
```
$ i=2
$ echo `expr 5 + \( $i \* 3 \)`
11
```

Expr Command: Basic Usage

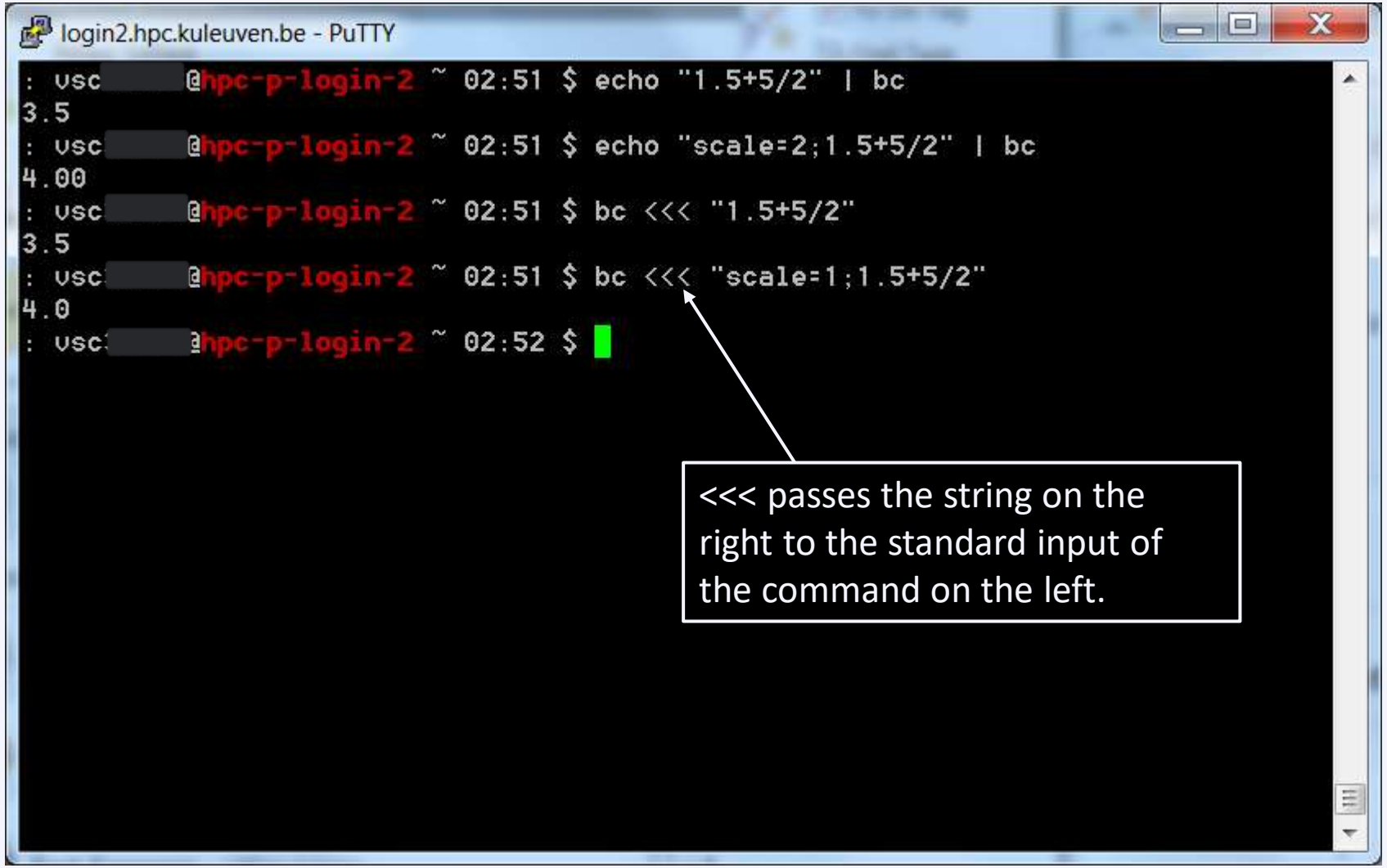
- Usage: expr EXPRESSION
  - `+` add
  - `–` subtract
  - `\*` multiply
  - `/` divide
  - `%` remainder after division (modulo operation)

- Examples (do not forget spaces!)
  - `expr 1 + 6` -> 7
  - `expr 2 \* 3` -> 6
  - `expr 4 % 3` -> 1

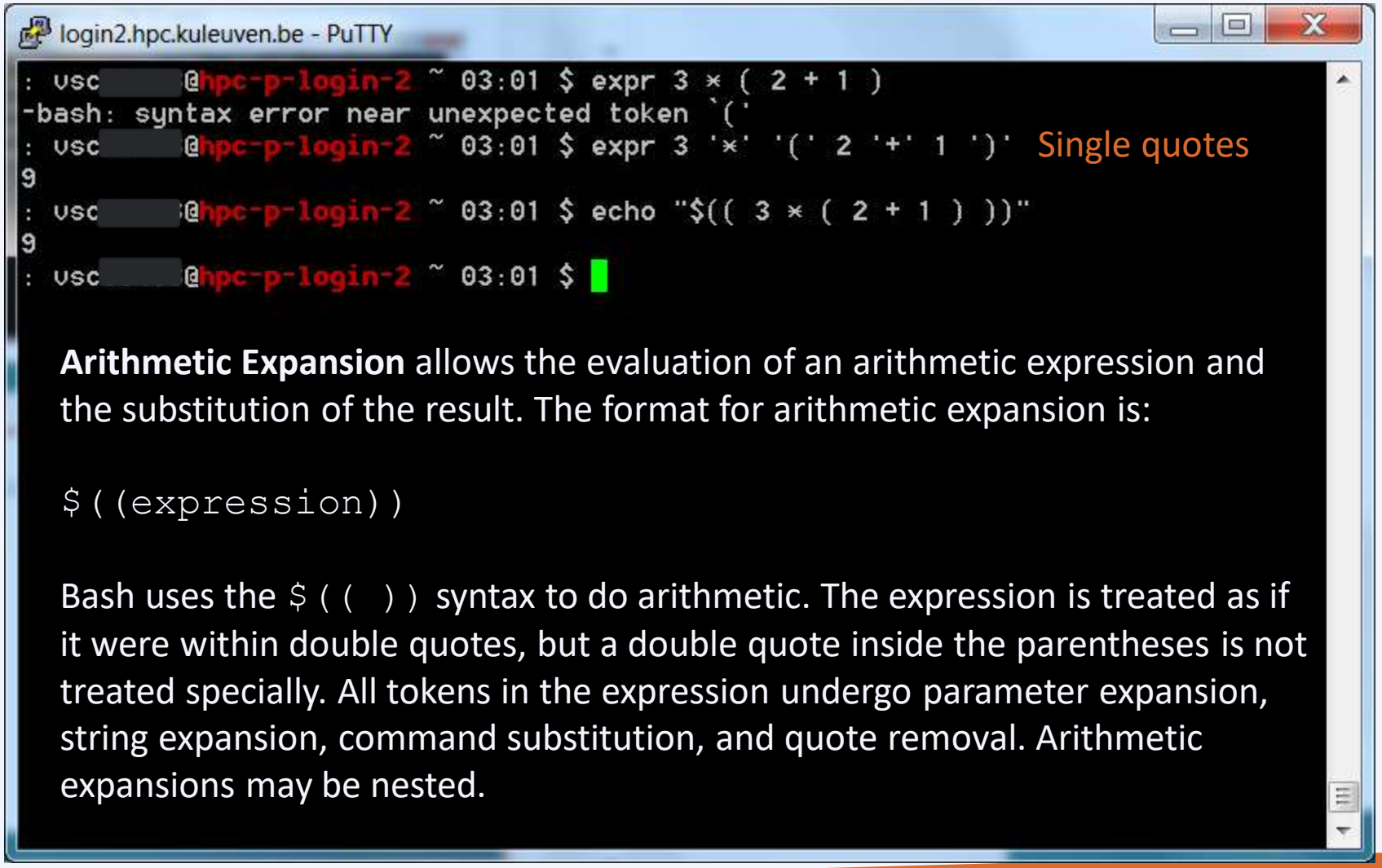
Example Of Expr



Alternative Of Expr



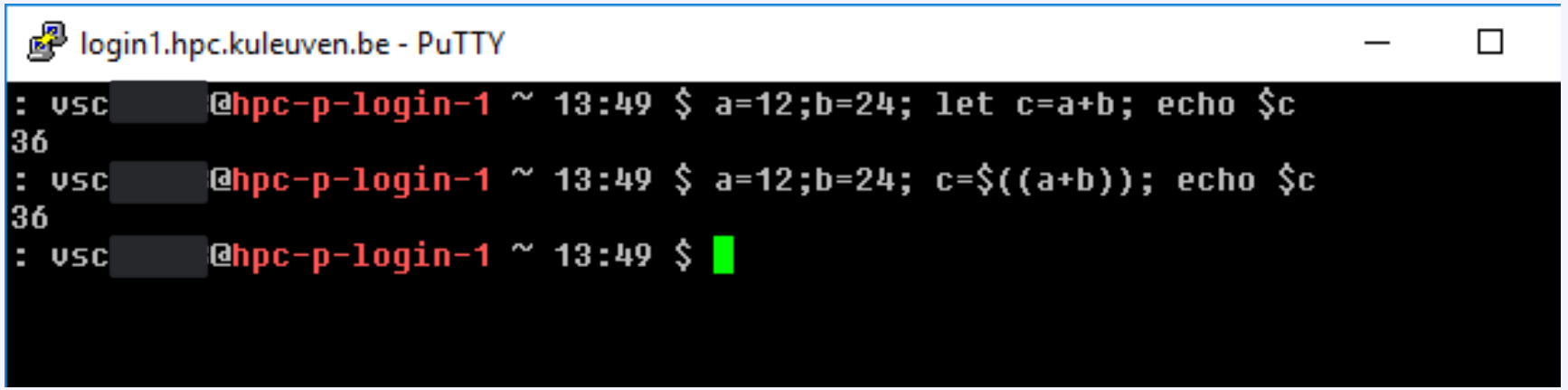
Complex Expr?



Let

`let` performs arithmetic on shell variables.
Note that each arithmetic expression has to be passed as a single argument to the let command, so you need quotes if there are spaces or globbing characters.
Let is very similar to ((but is simpler way to do arithmetic operations.

```
$ a=12;b=24; let c=a+b; echo $c
36
```



Let

`let "a += 5"` # Equivalent to `let "a = a + 5"` (Double quotes and spaces make it more readable.)

`let "a /= 4"` # Equivalent to `let "a = a / 4"`

`let "a -= 5"` # Equivalent to `let "a = a - 5"`

`let "a *= 10"` # Equivalent to `let "a = a * 10"`

`let "a %= 8"` # Equivalent to `let "a = a % 8"`

Arithmetic operators

Operator	Description	Example	Result
+	Addition	`echo $(( 7 + 5 ))`	12
-	Subtraction	`echo $(( 7 - 5 ))`	2
/	Division	`echo $(( 6 / 3 ))`	2
*	Multiplication	`echo $(( 2 * 3 ))`	6
%	Modulo	`echo $(( 5 % 3 ))`	2
**	Exponentiation	`echo $(( 2 ** 3 ))`	8

- `$RANDOM` is an internal Bash function (not a constant)
  that returns a pseudorandom integer in the range 0 - 32767

- How to generate an inclusive random number between 1 to 10
  `r=$(( RANDOM % 10 + 1 )); echo $r`

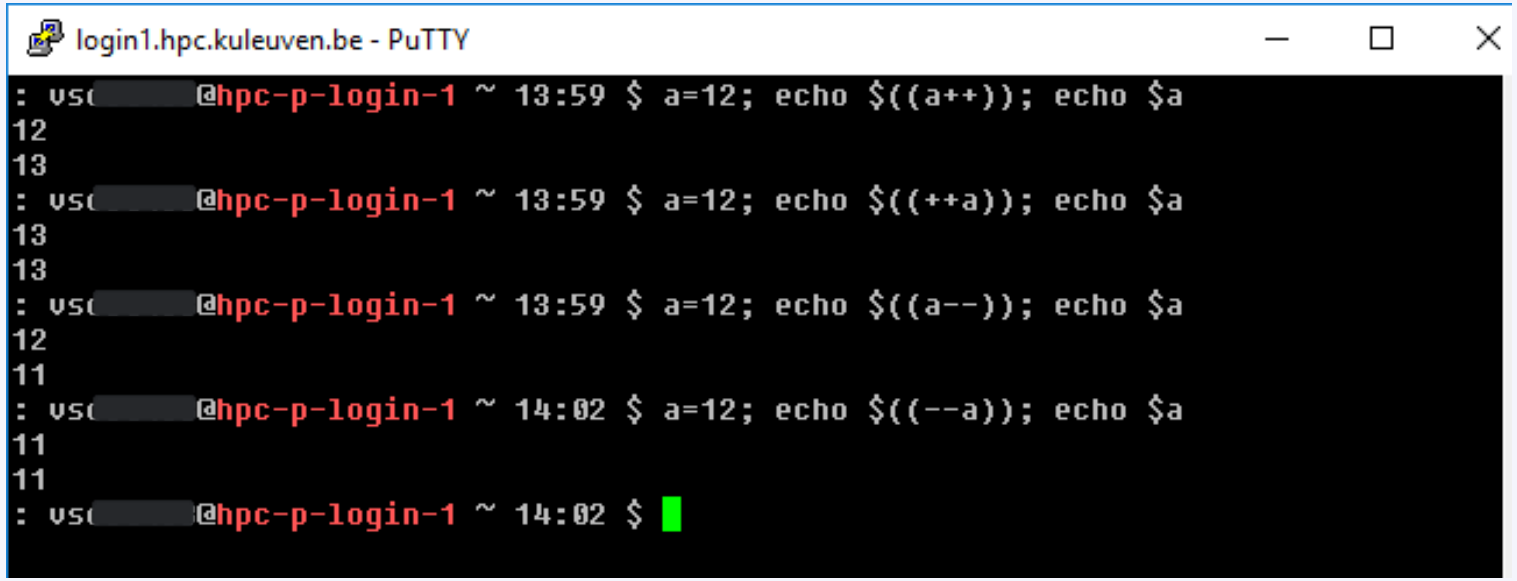
Pre/Post Increment/Decrement

`x++` post-increment
`x--` post-decrement
`++x` pre-increment
`--x` pre-decrement

- The *post*-increment and *post*-decrement operators increase (or decrease) the value of their operand by 1, but the value of the expression is the operand's original value *prior* to the increment (or decrement) operation.
- The *pre*-increment and *pre*-decrement operators increment (or decrement) their operand by 1, and the value of the expression is the resulting incremented (or decremented) value.

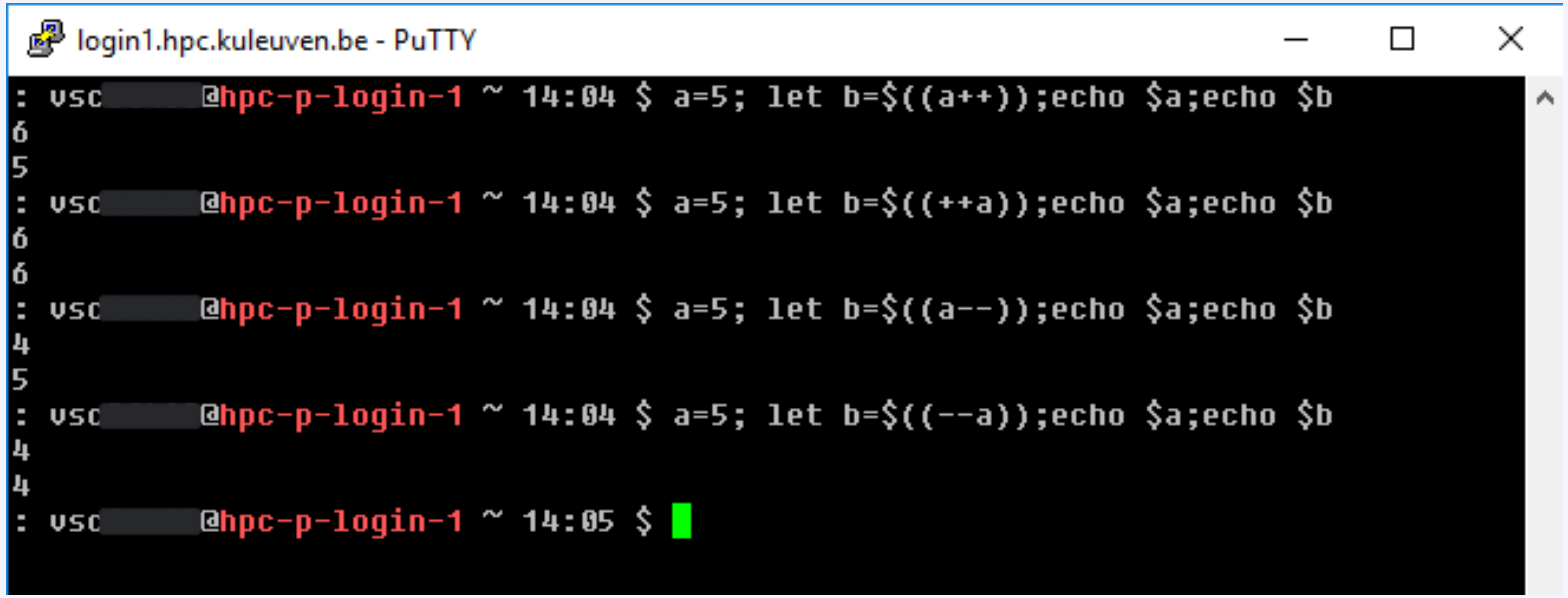
Pre/Post Increment/Decrement

`x++` post-increment – makes a copy, increases x, returns the copy (old value)
`x--` post-decrement – makes a copy, decreases x, returns the copy (old value)
`++x` pre-increment – increases x, and returns x (new value)
`--x` pre-decrement – decreases x, and returns x (new value)



Pre/Post Increment/Decrement

`x++` post-increment
`x--` post-decrement
`++x` pre-increment
`--x` pre-decrement



Other expr Options

- `|`
  - Logical or
- `&`
  - Logical and
- Many others… check the man page
  - `ARG1 < ARG2` ARG1 is less than ARG2
  - `ARG1 <= ARG2` ARG1 is less than or equal to ARG2
  - `ARG1 = ARG2` ARG1 is equal to ARG2
  - `ARG1 != ARG2` ARG1 is unequal to ARG2
  - `ARG1 >= ARG2` ARG1 is greater than or equal to ARG2
  - `ARG1 > ARG2` ARG1 is greater than ARG2

Control Flow And Conditionals

- Shell scripts can be more powerful than just a list of commands
- Sometimes you want to perform some commands only if certain conditions are true
  - Example: Only print out a file if it is not a binary file

The test Command

- Used to check if certain conditions are true
- Usage: `test EXPRESSION`
- You get 0 (true) or 1 (false) in the exit code (`echo $?`).

Conditions test Checks

- Comparisons
  - `-eq`
  - `-ne`
  - `-lt`
  - `-le`
  - `-gt`
  - `-ge`

More Conditions

- System conditions
  - `-d`
    - File is a directory: `test -d $VSC_HOME`
  - `-e`
    - File exists: `test -e ~/.bashrc`
  - `-f`
    - File is a normal file (not a directory, device)
  - `-s`
    - File is non-empty
  - `-r`
    - True if ‘file’ is readable
  - `-w`
    - True if ‘file’ is writable
  - `-x`
    - True if ‘file’ is executable

More test Conditions

- `!`
  - Not
  - Negates the next check
  - Example: `test ! -x File`
- `-a`
  - And two conditions
    - Example: `test $1 -eq $2 -a $2 -gt 5`
- `-o`
  - Or two conditions
    - Example: `test $1 -eq $2 -o $2 -gt 5`

Relational operators

Meaning	Numeric	String
Greater than	`-gt`	
Greater than or equal	`-ge`	
Less than	`-lt`	
Less than or equal	`-le`	
Equal	`-eq`	`= or ==`
Not equal	`-ne`	`!=`
str1 is less than str2		`str1 < str2`
str1 is greater str2		`str1 > str2`
String length is greater than zero		`-n str`
String length is zero		`-z str`

Shortcut For Test

- Because this is used very often, a shortcut exists
- `[]`
  - Example: `[ -f File ]`
  - `test -f File`

When Do You Use This?

- `test` is used to control the operation of your script
- The answer from test should guide the execution of your code one way or another
- Used in conditional statements

Logic: test

```
$ test 1 -lt 10
$ echo $?
0
$ test 1 == 10
$ echo $?
1
```

Logic: test

- `test`
- `[ ]`
  - `[ 1 -lt 10 ]`
- `[[ ]]`
  - `[[ “this string” =~ “this” ]]`
- `(( ))`
  - `(( 1 < 10 ))`

Notice the mandatory space between brackets and the arguments

The =~ Regular Expression matching operator within a double brackets test expression.

Logic: test

- Number (arithmetic comparisons):
  - `-eq`
  - `-ge`
  - `-le`
  - `-ne`
  - `-gt`
  - `-lt`
- Use with `[ ]`
  - `[ $var1 -lt $var2 ]` -> true if var1 less than var2 , else false

Logic: test

- Comparisons operators
  - `>`
  - `>=`
  - `<`
  - `<=`
- Use with `(( ))`
  - `(( $var1 < $var2 ))` -> true if var1 less than var2 , else false

Logic: test

- `[ -f /etc/passwd ]`
- `[ ! -f /etc/passwd ]`
- `[ -f /etc/passwd -a -f /etc/shadow ]`
- `[ -f /etc/passwd -o -f /etc/shadow ]`

Key concepts

- Multiple commands can be separated with a **;**

  e.g. `cd $VSC_HOME;pwd`

- **&&** and **||** conditionally separate multiple commands. When commands are conditionally joined, the first will always execute. The second command may execute or not, depending on the return value of the first command. For example, a user may want to create a directory, and then move a new file into that directory. If the creation of the directory fails, then there is no reason to move the file. The two commands can be coupled as follows:
  - `echo „one two three four five” > numbers.txt;`
  - `mkdir /tmp/my-dir && mv numbers.txt /tmp/my-dir`

Key concepts

- Similarly, multiple commands can be combined with **||**. In this case, **bash** will execute the second command only if the first command "fails" (has a non zero return value). This is similar to the "or" operator found in programming languages. In the following example, we attempt to change the permissions on a file. If the command fails, a message to that effect is echoed to the screen.
  - `chmod 600 /tmp/my-dir/numbers.txt || echo ”chmod failed”`
  - `chmod 600 /tmp/my-dir/Numbers.txt || echo “chmod failed”`
- In the first case, the **chmod** command succeeded, and no message was echoed. In the second case, the **chmod** command failed (because the file didn't exist), and the "chmod failed" message was echoed (in addition to **chmod**'s standard error message).

Key concepts: escape character

- ", $, `, and \ are still interpreted by the shell, even when they're in double quotes.
- The backslash (\) character is used to mark these special characters so that they are not interpreted by the shell, but passed on to the command being run (for example, echo)
- E.g. to output the string: (Assuming that the value of $X is 5):

```
A quote is ", backslash is \, backtick is `.
A few spaces are   and dollar is $. $X is 5.
```

we would have to write:

```
$ echo "A quote is \", backslash is \\, backtick is \`."
A quote is ", backslash is \, backtick is `.

$ echo "A few spaces are   ; dollar is \$. \$X is ${X}."
A few spaces are   ; dollar is $. $X is 5.
```

Key concepts: escape character

- $ is used for interpreting a variable which has some value assigned

- When you create a file that contains space in it, e.g. `touch "my file"` it is difficult to use it later
  - How to copy the file (`cp source destination`)
  - -> use escape character so that space is understood as a part of the file and not as a separator in comand syntax
  - `cp my\ file myfile`

- Better avoid using „special" characters (", $, `, \, …) in your filenames!

Key concepts: history

- ! is used in history event designators: you can execute a previous command using `![N]` where N is the line number in history you want to recall (or 2 commands before `!-2`)
- `echo "Hi $USER!"` works from the script, but gives an error "`-bash: !": event not found`" from interactive bash shell (because of history)
- There are 2 ways to deal with that:
  - `echo "Hi $USER"'!'`
  - `$ set +H` - switch off `-H` option in bash (`-H` is used to enable `!` style history substitution. This option is on by default when the shell is interactive). To set it back use `$ set -H`

Hands-on 2

1. Write the script that reads integer input and displays the result *Initial value is:* `value`. Next it adds 2 to the read value and displays the result *Value after adding is* `value`. After that it multiplies the result by 3 and displays the new result *Value after multiplying is* `value`. Finally, it calculates modulo 2 and displays the result as *Value after performing modulo is* `value`. Run the script.
2. Try to run the script for input parameter that is not integer, but real, e.g. 1.5. What happens?
3. Correct the script so that it does the same for real parameters. Use the `-n` option for echo to skip printing newline at the end.
4. Write the script that reads 2 integer parameters and checks if the first value is greater or equal than the second value. Display the exit code. Test the script with different values.
5. Modify the script such that it tests if the first value is greater than the second one. If test is succeeded it should display "*True*", if test fails it should display "*False*".

Shell logic structures

The four basic logic structures needed for program development are:

- Sequential logic: to execute commands in the order in which they appear in the program
- Decision logic: to execute commands only if a certain condition is satisfied
- Looping logic: to repeat a series of commands for a given number of times
- Case logic: to replace “if then/else if/else” statements when making numerous comparisons

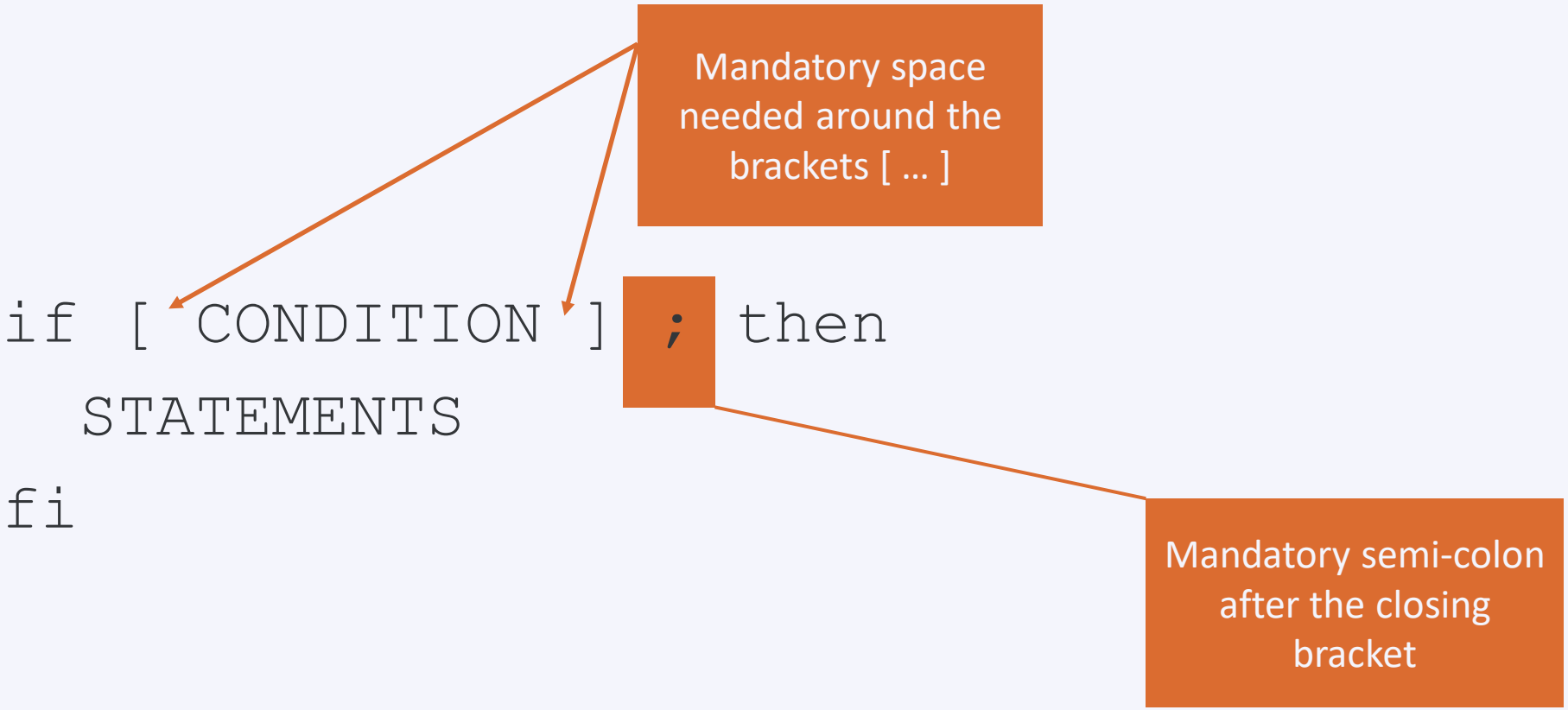
Loops

Loop is a block of code that is repeated a number of times.

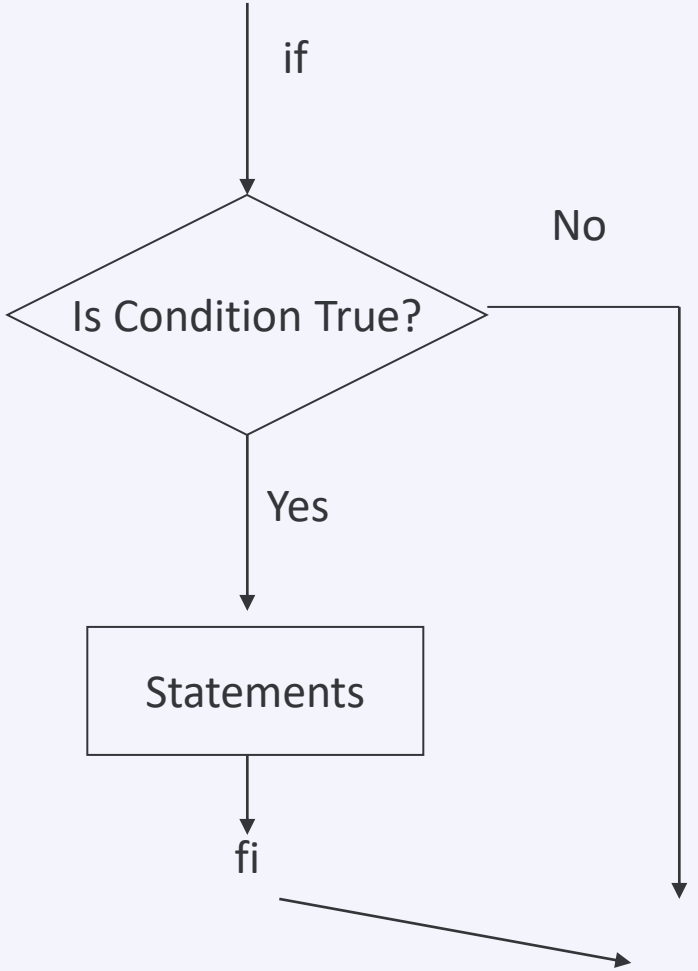
The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (for loops) or until a particular condition is satisfied (while and until loops)

To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

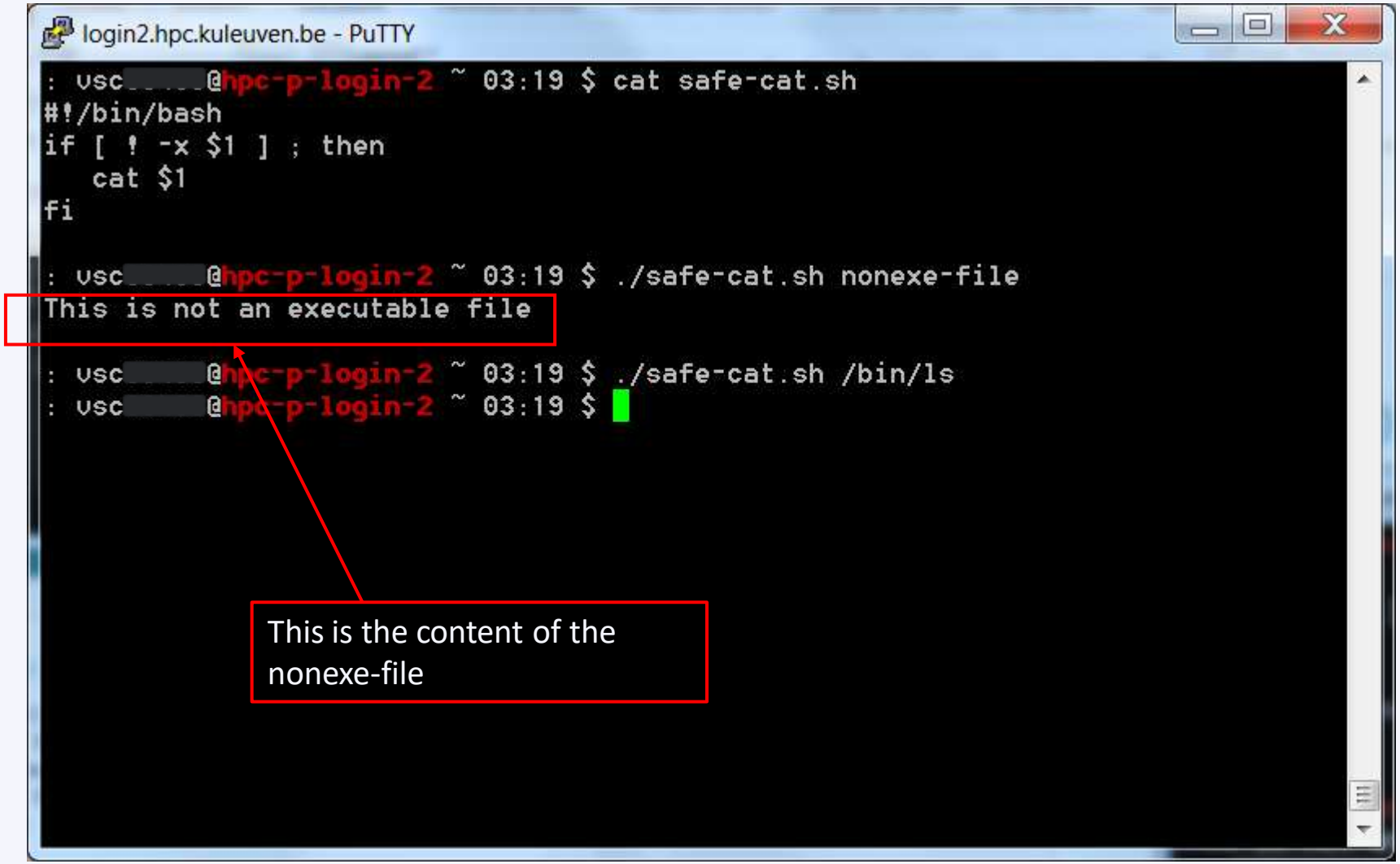
if then fi


Mandatory space needed around the brackets […]
if [CONDITION] ; then
STATEMENTS
fi
Mandatory semi-colon after the closing bracket

Flowchart of if



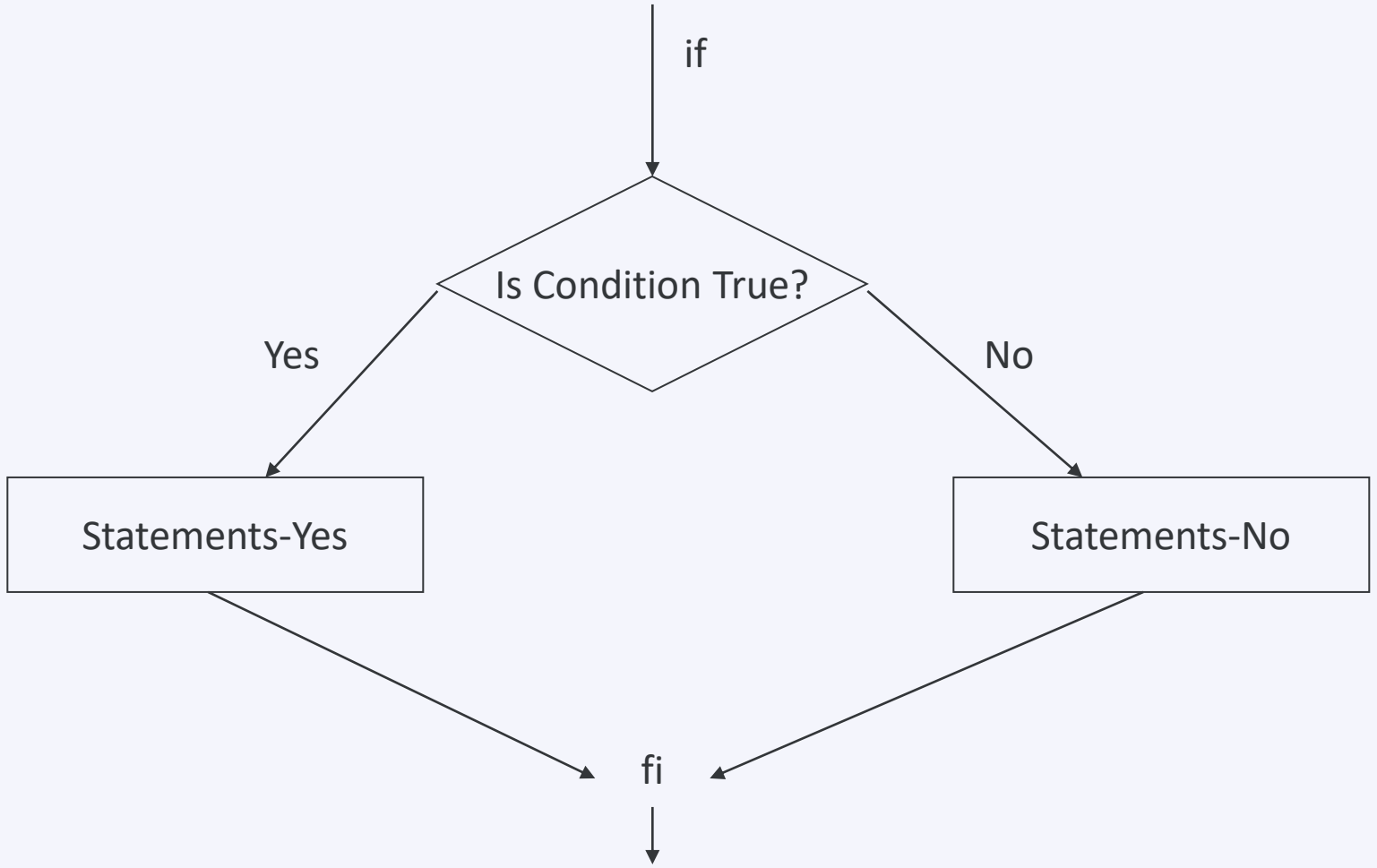
Example of if



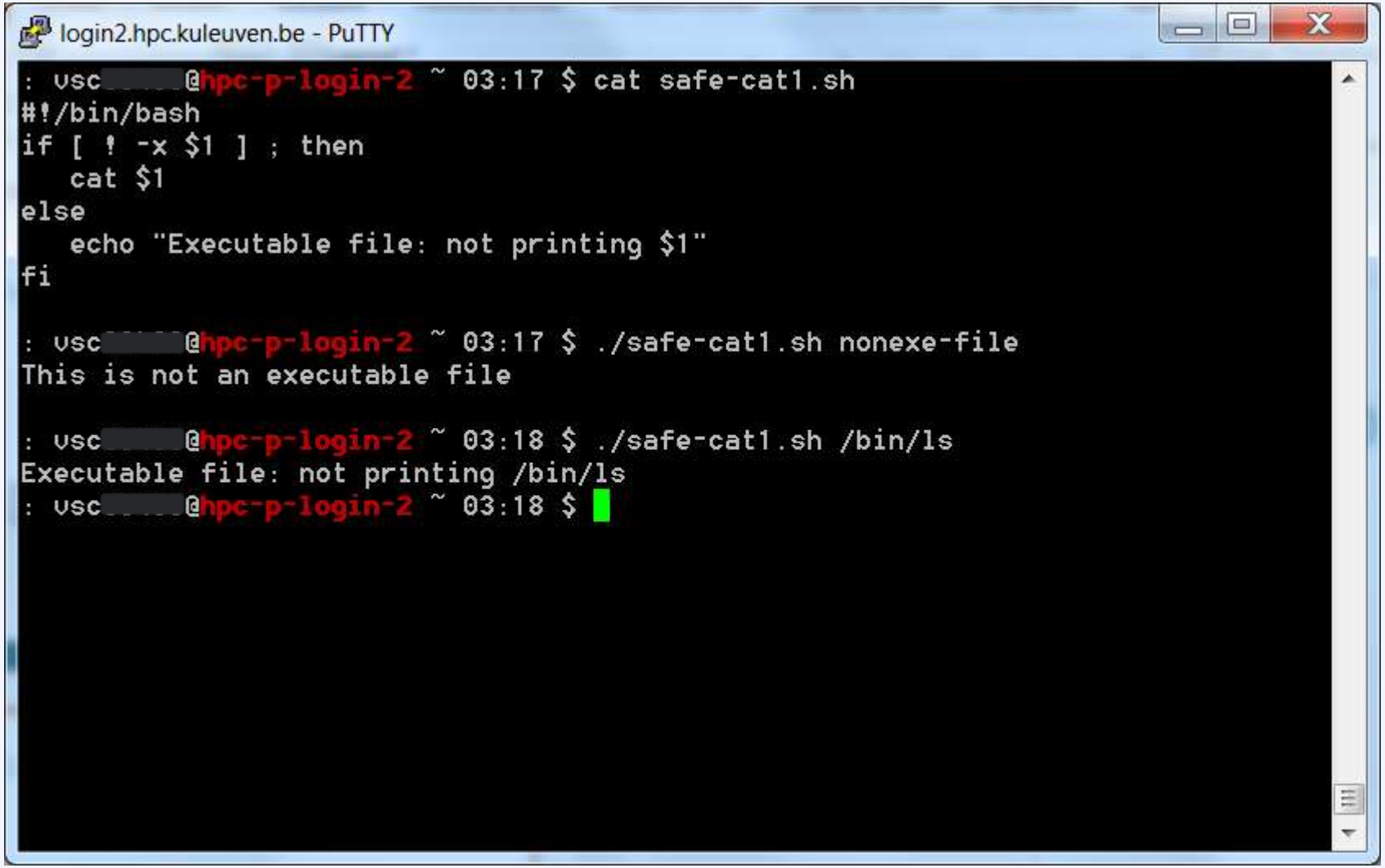
If then else fi

```
if [ CONDITION ] ; then
   STATEMENTS-YES
else
   STATEMENTS-NO
fi
```

Flowchart of if else



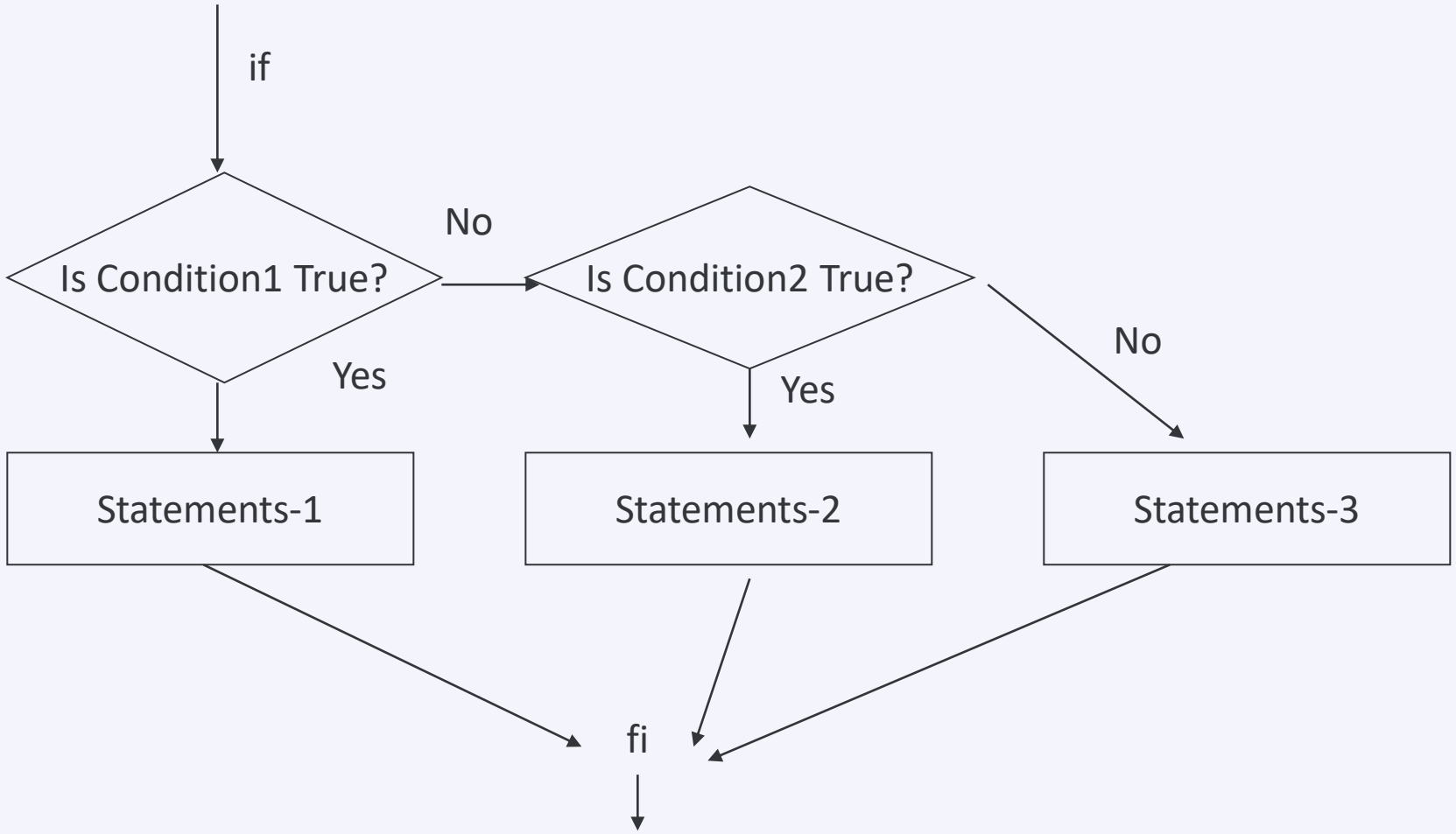
Example of if else



if then elif else fi

```
if [ CONDITION1 ] ; then
   STATEMENTS-1
elif [ CONDITION2 ] ; then
   STATEMENTS-2
else
   STATEMENTS-3
fi
```

Flowchart of if elif else



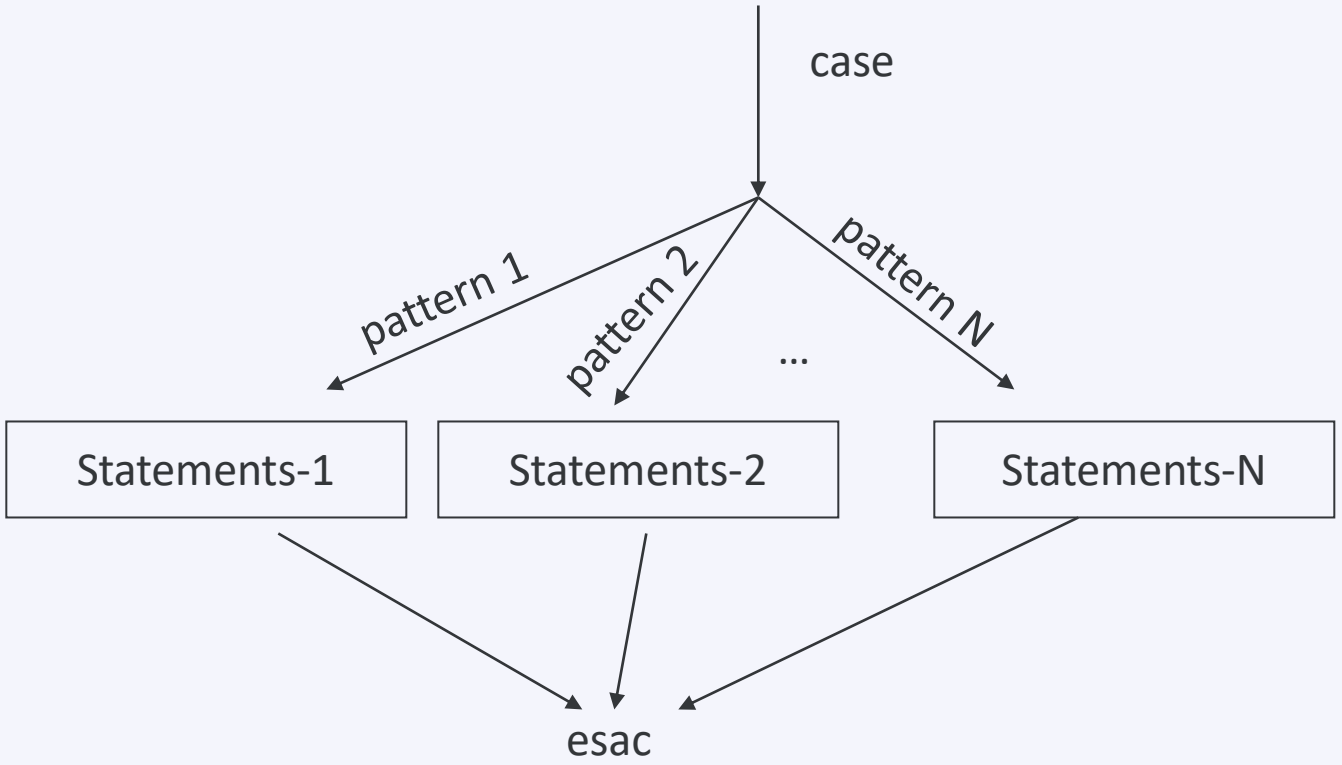
case; esac

```
case STRING in
pattern1)
            STATEMENTS-1 ;;
pattern2)
            STATEMENTS-2 ;;
…
patternN)
            STATEMENTS-N ;;
esac
```

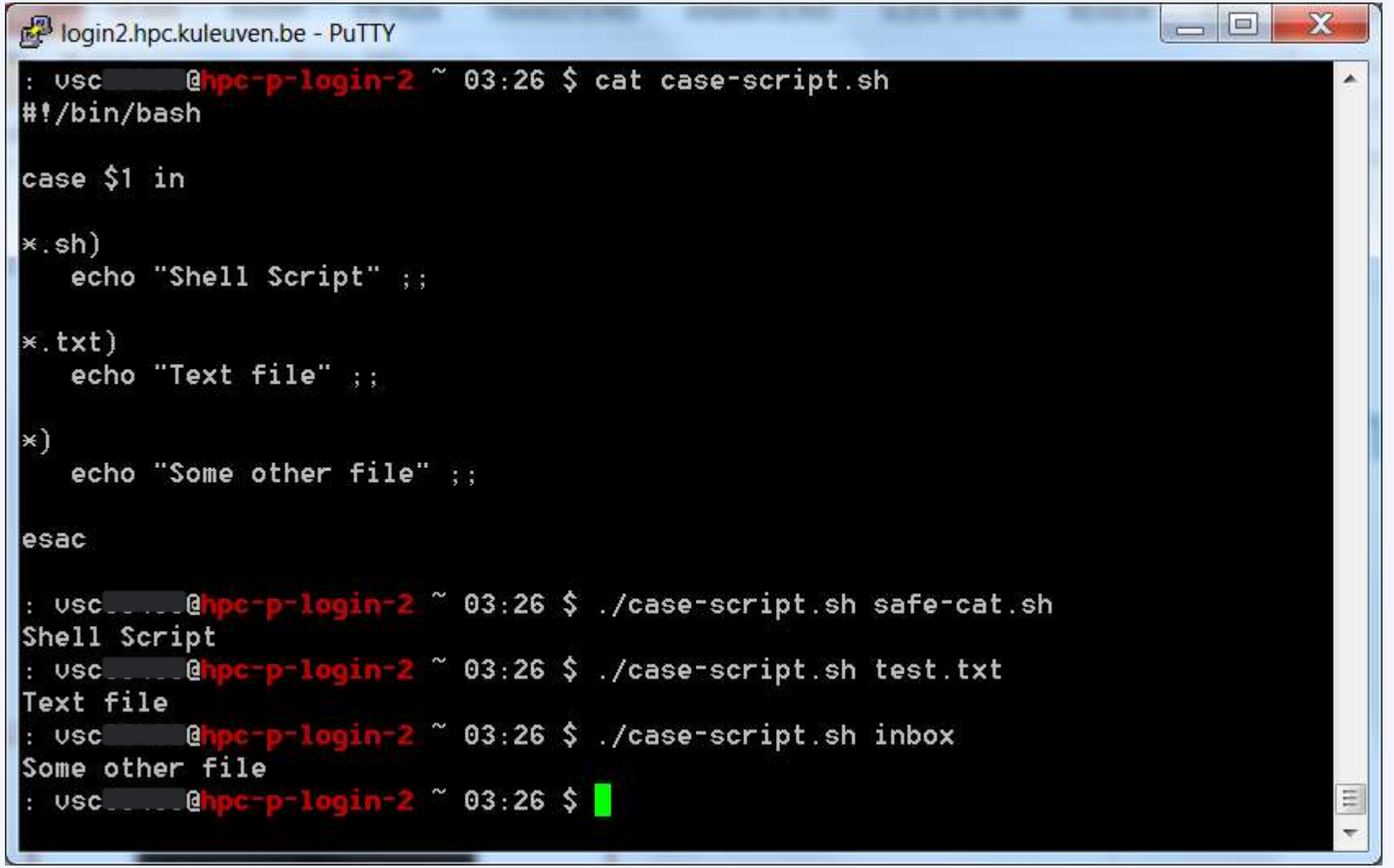
Note: right-sided parenthesis

Note: Two semi-colons ;; at the end of each statement group

Flowchart of case



Example of case



Universal customization

Universal .bashrc - written to run on all (relevant) clusters:

```
case ${VSC_INSTITUTE_CLUSTER} in
    wice)
        export PS1=': \u@\[\e[1;31m\]\h\[\e[0m\] \w `date +%H:%M` $'
        ;;
    genius)
        echo "Genius has Skylake and CascadeLake partitions"
        export EDITOR="/usr/bin/vim"
        export PS1=': \u@\[\e[1;34m\]\h\[\e[0m\] \w `date +%H:%M` $ '
        ;;
esac
```

select

- **Use 1: Creating a simple menu**
- **`select`** command will retrieve each data from the list and print the data as a menu.
- After executing the script it will create a menu and ask the user to choose anything. It will print the name of the selected case.
- The user has to press **Ctrl+c** to terminate from the script unless break statement is used.

select

```
select ITEM in [LIST]
do
  [COMMANDS]
done
```

- The `[LIST]` can be a series of strings separated by spaces, a range of numbers, output of a command, an array, and so on.
- When the `select` construct is invoked, each item from the list is printed on the screen (standard error), preceded with a number.

select

```
select ITEM in [LIST]
do
  [COMMANDS]
done
```

- If the user enters a number that corresponds to the number of one of the displayed items, then the value of `[ITEM]` is set to that item. The value of the selected item is stored in the variable `REPLY`.

select

The user has to press **`Ctrl+c`** to terminate from the script or **`break`** has to be used.

```
[Oct/19 22:57] vsc[redacted]@tier2-p-login-3 /vsc-hard-mounts/leuven-user/[redacted]/vs[redacted]
[redacted]/course/scripting $ more select.sh
#!/bin/bash
# Define the menu list here
select cluster in Genius wICE Superdome BrEniac Hortense
do
echo "You have chosen $cluster"
break
done

[Oct/19 22:58] vsc[redacted]@tier2-p-login-3 /vsc-hard-mounts/leuven-user/[redacted]/vs[redacted]
[redacted]/course/scripting $ ./select.sh
1) Genius
2) wICE
3) Superdome
4) BrEniac
5) Hortense
#? 1
You have chosen Genius
[Oct/19 22:58] vsc[redacted]@tier2-p-login-3 /vsc-hard-mounts/leuven-user/[redacted]/vs[redacted]
[redacted]/course/scripting $
```

select

- **Use 2: Select command with a case statement**

- After running the script, the user will select any menu item and case statement will match the selected value with case value.

- Here multiple case values can be used for matching with the selected menu item. If none of the case value matches with the selected menu item then “Invalid entry” will print.

select

```
    /course/scripting $ more select-case.sh
#!/bin/bash
echo "Which Operating System do you like?"
select os in Ubuntu CentOS Windows10 Windows11
do

case $os in
# Two case values are declared here for matching
"Ubuntu"|"CentOS")
echo "I also use $os."
;;
# Three case values are declared here for matching
"Windows11" | "Windows10")
echo "Why don't you try Linux?"
;;
# Matching with invalid data
*)
echo "Invalid entry."
break
;;
esac
done
 [Oct/19 23:06] vsc      @tier2-p-login-3 /vsc-hard-mounts/leuven-user/    /vs
    /course/scripting $ ./select-case.sh
Which Operating System do you like?
1) Ubuntu
2) CentOS
3) Windows10
4) Windows11
#? 2
I also use CentOS.
#? 3
Why don't you try Linux?
#? 5
Invalid entry.
 [Oct/19 23:07] vsc      @tier2-p-login-3 /vsc-hard-mounts/leuven-user/    /vs
    /course/scripting $
```

Hands-on 3

1. Write the script that performs summation of two integers. Before adding the numbers introduce the `if` statement that checks if 2 parameters were given. Test it with 2 and more parameters given.

2. Write the script that finds the biggest number of 3 given integer numbers Use `if - elif` block multiple times.

3. Add the check in the beginning of the script that detects faulty execution and instructs how to use the script, e.g. *“Use biggest.sh: number1 number2 number3”*. Test it with 2 and 3 parameters.

4. Add extra checks to the script that detects if all the values are the same and displays *"All the three numbers are equal"* and the information printed if the values are not integer values that can be compared *"I can not figure out which number is bigger"*

5. Write the script that performs 4 operations (summation, subtraction, multiplication and division) of two given integer numbers and the operator. Use `case` statement.

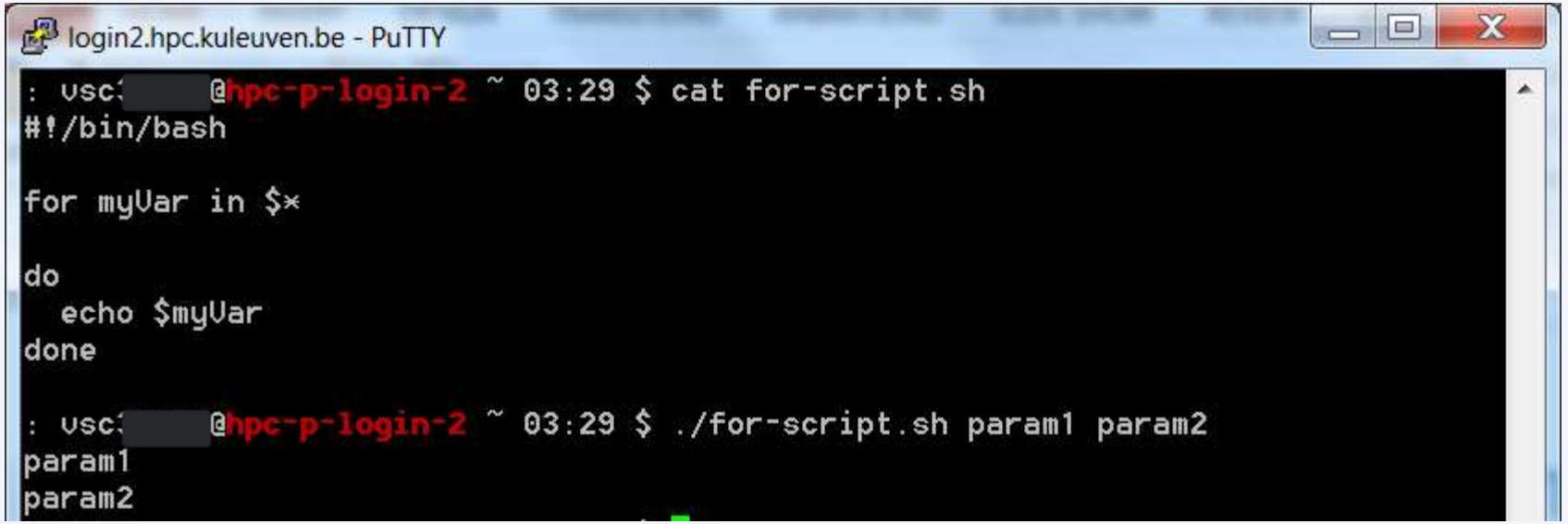
Looping

- Sometimes you want to do something many times
- Loop for a set number of times
- Loop while a condition is true
- Loop until a condition is false

for loop

Syntax

```
for VAR in LIST
do
   STATEMENTS
done
```



Special form of for loop

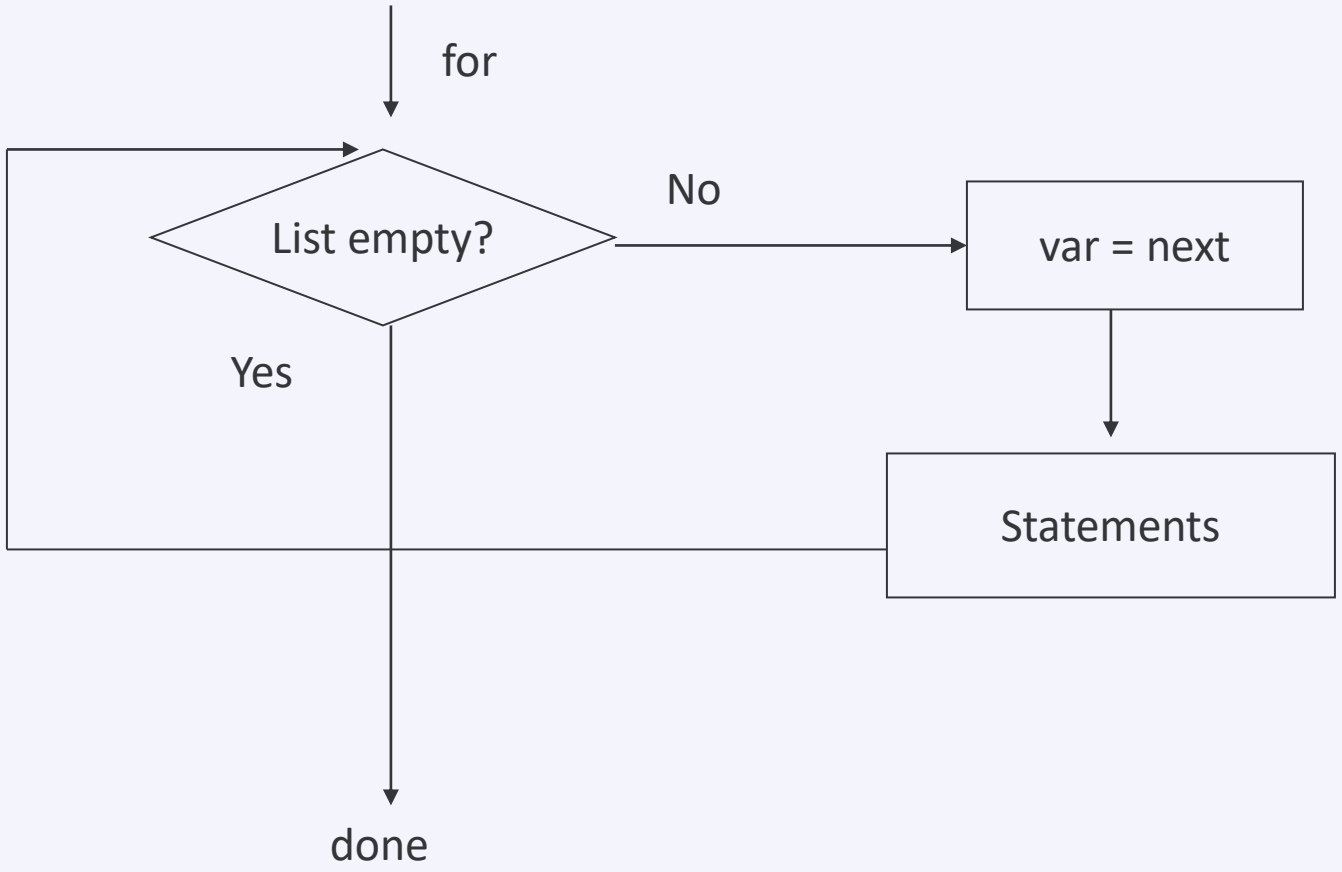
Syntax

Without a list, it will go through all of the command line arguments

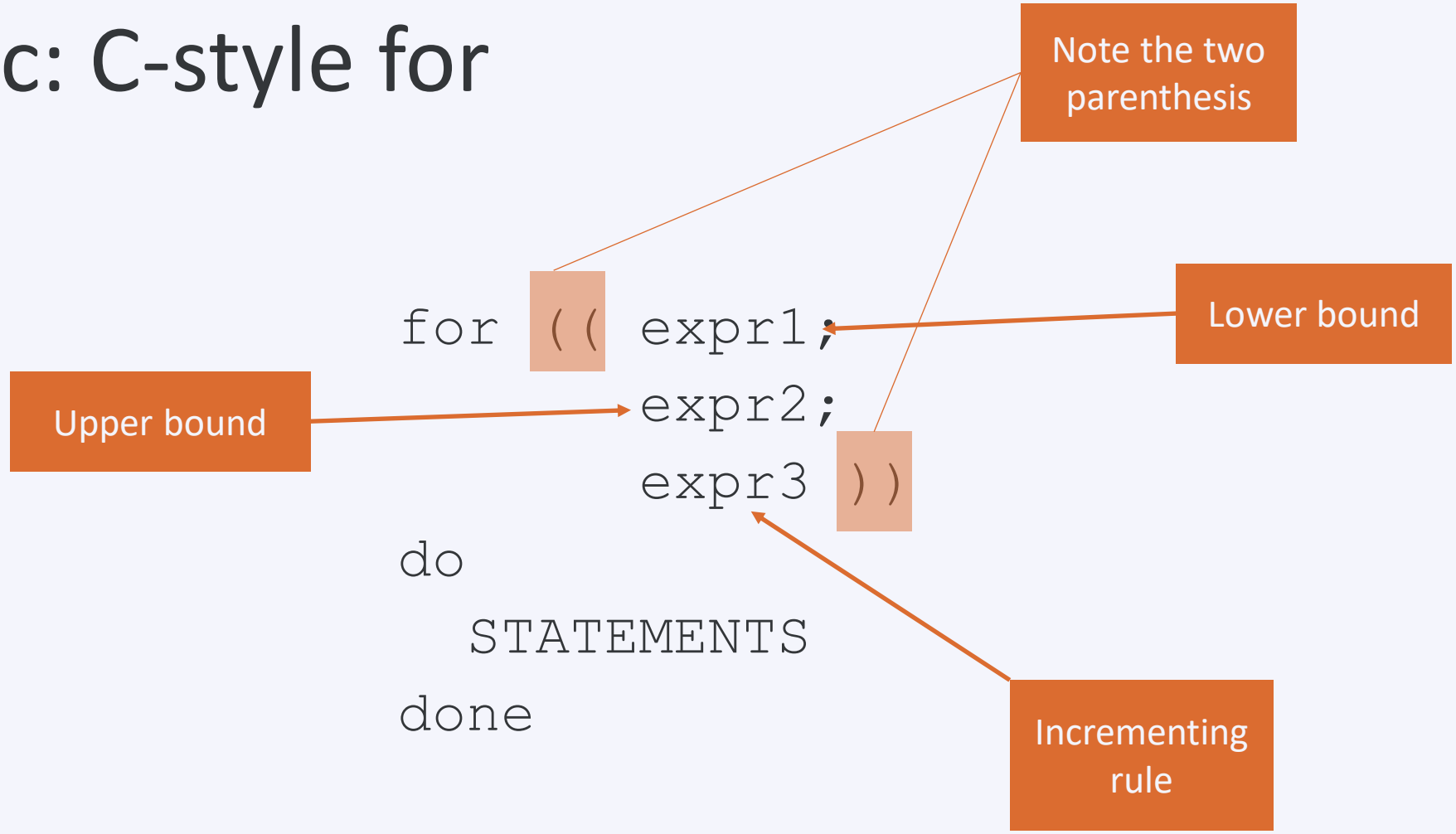
```
for var
do
    STATEMENTS
done
```

```
ehsan@CRD-L-      :~$ cat for-loop.sh
#!/bin/bash
for var
do
    echo $var
done
ehsan@CRD-L-      :~$ ./for-loop.sh param1 param2
param1
param2
ehsan@CRD-L-      :~$
```

Flowchart of for loop

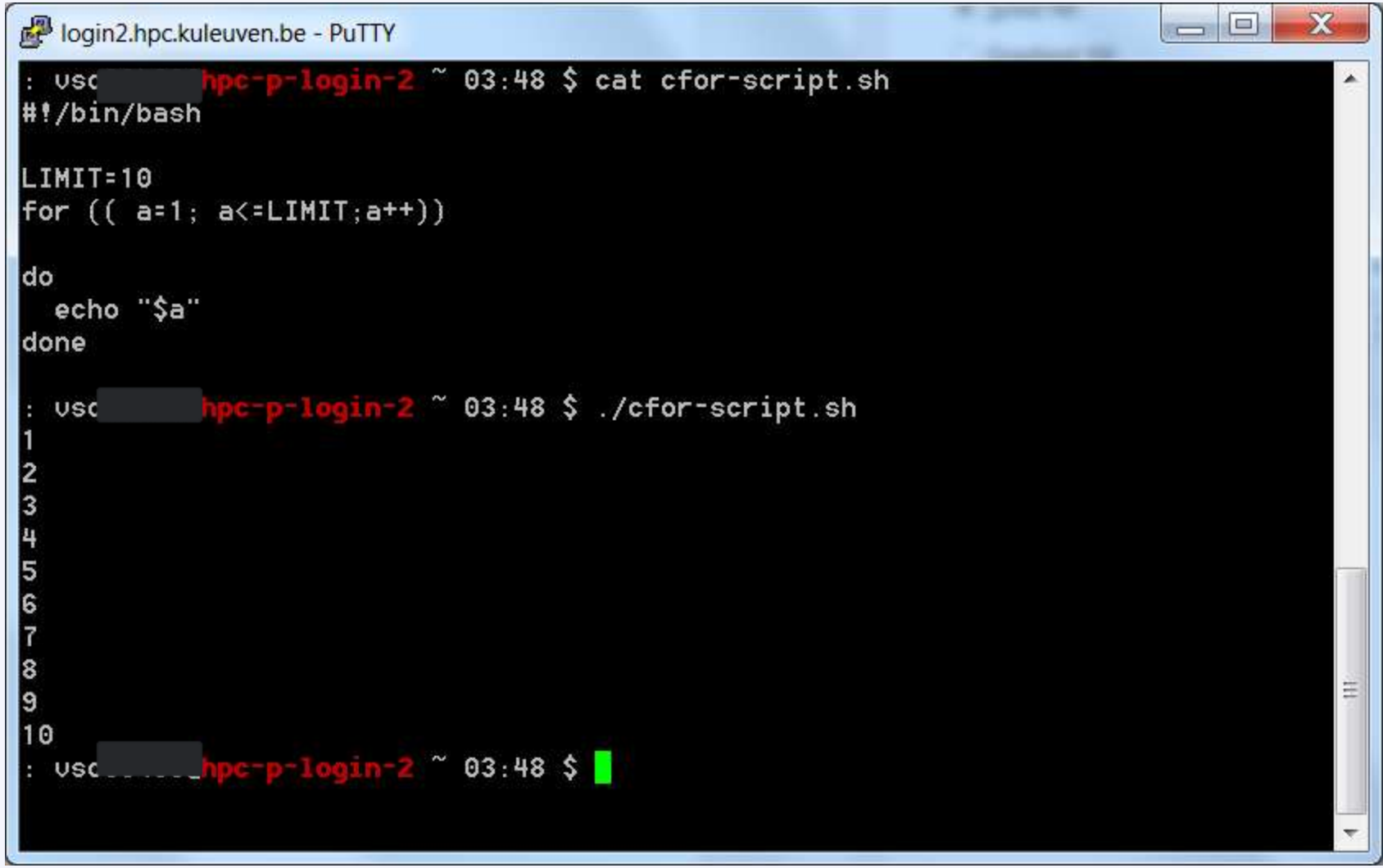


Logic: C-style for



Logic: C-style for

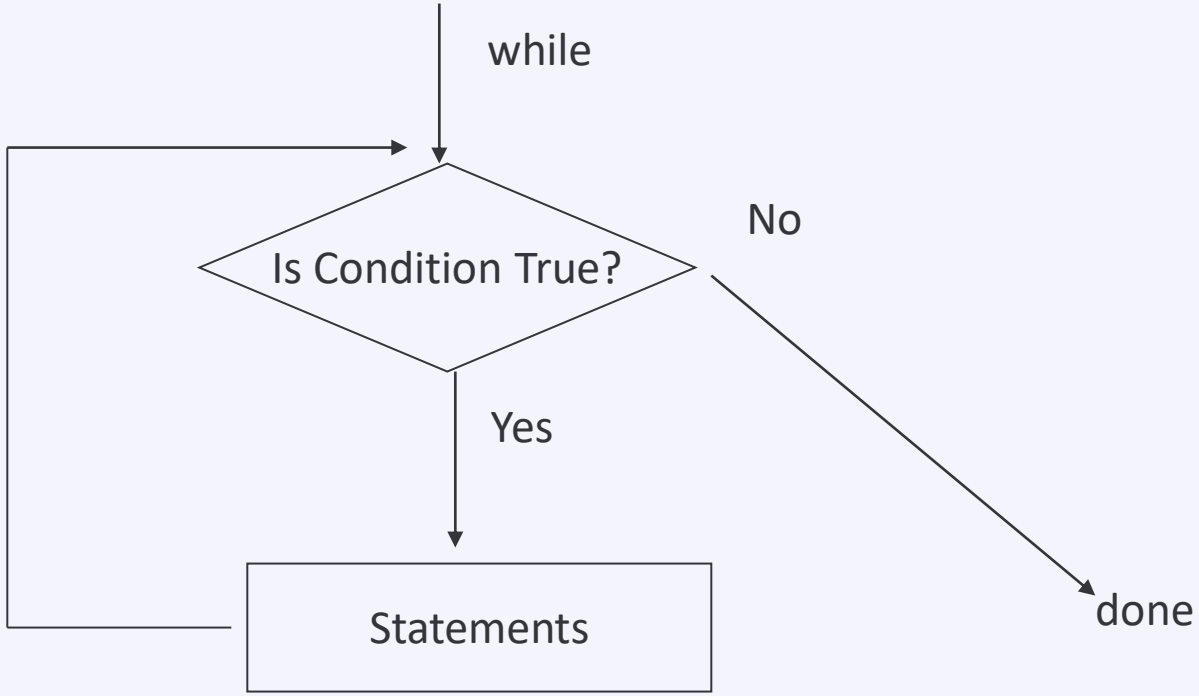
```
LIMIT=10
for (( a=1          ;
       a<=LIMIT     ;
       a++          ))
do
   echo -n “$a ”
done
```



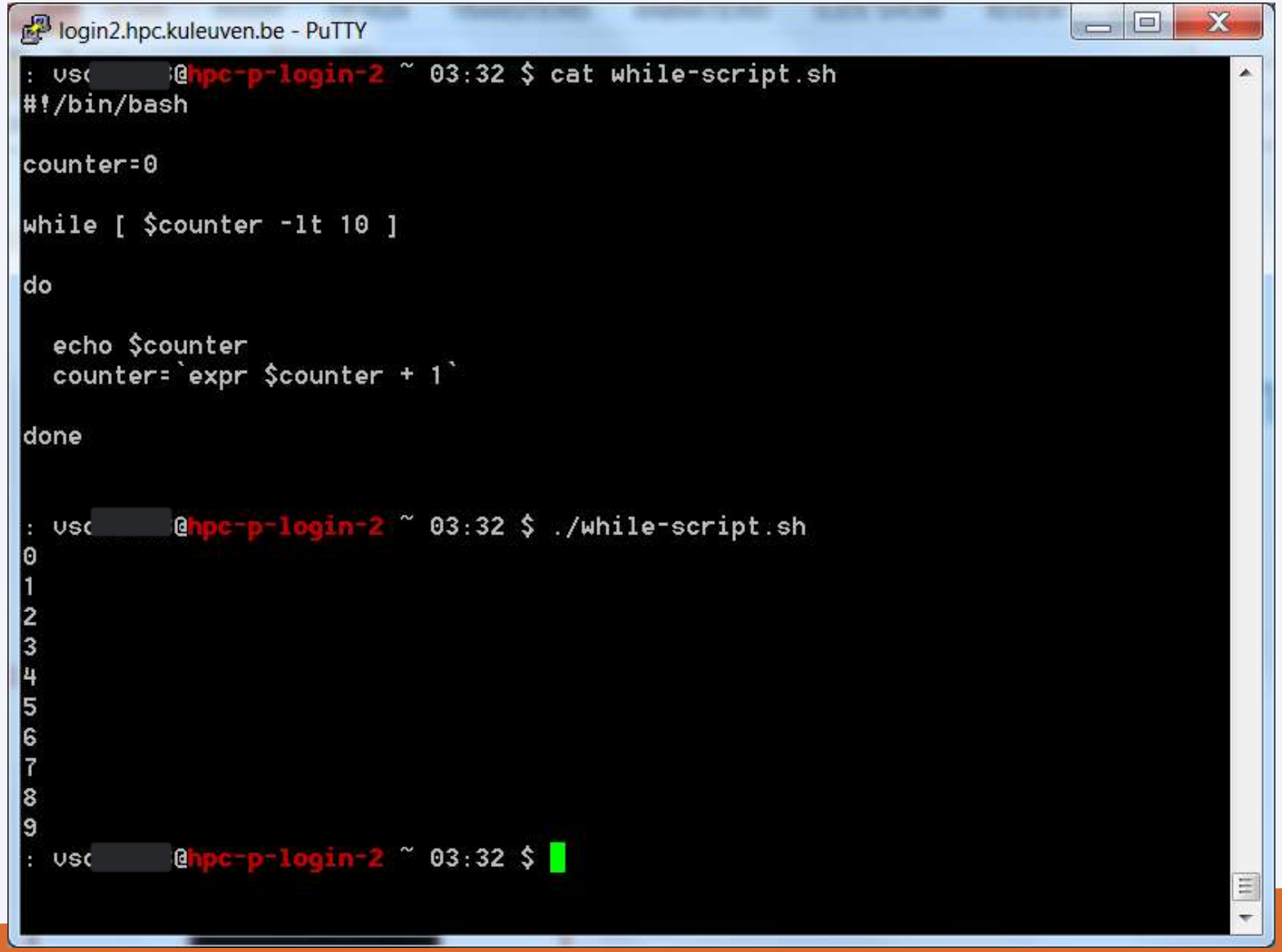
While loop

```
while CONDITION
do
      STATEMENTS
done
```

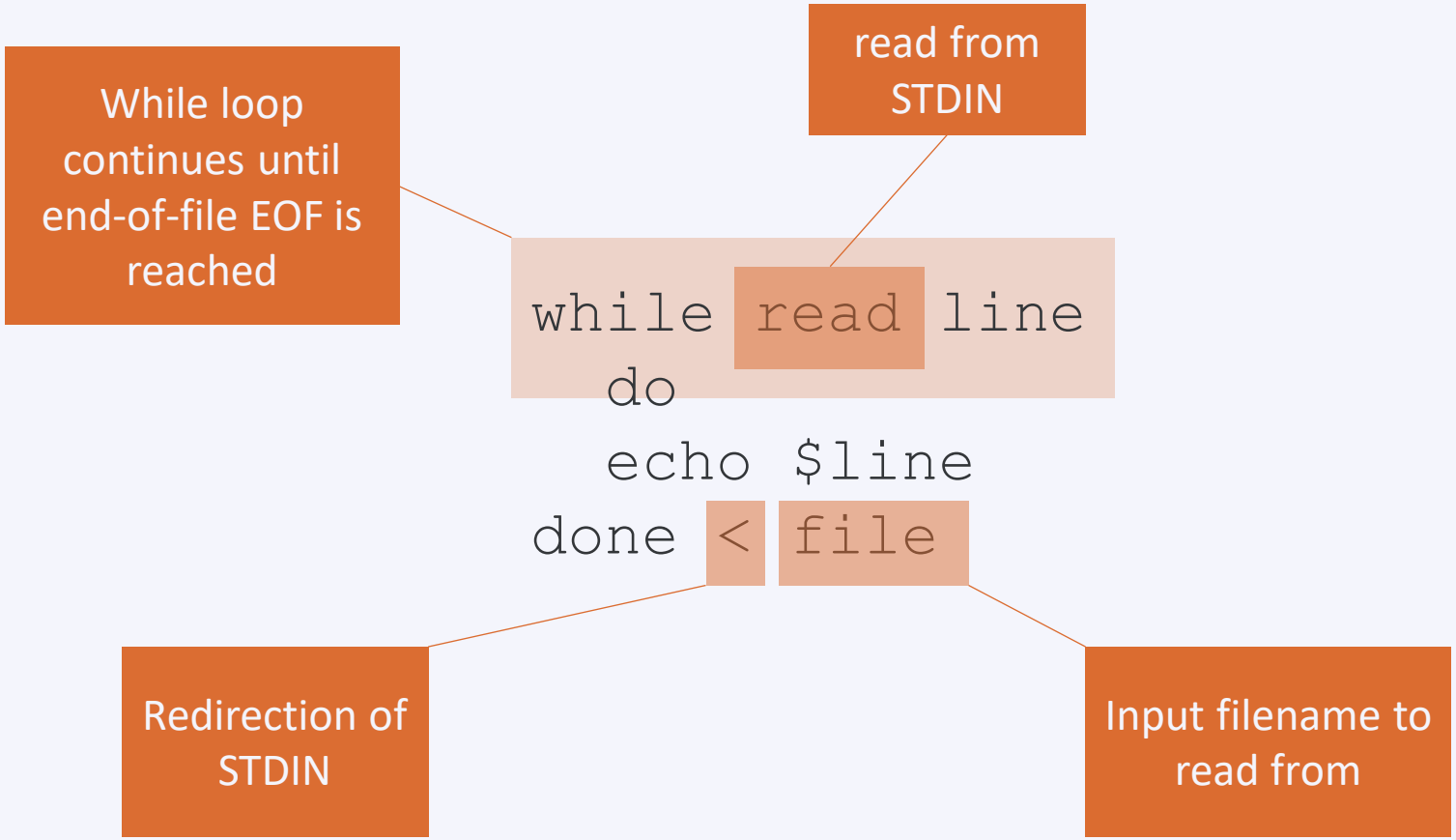
Flowchart of while loop



Example of while loop



while loop – reading files

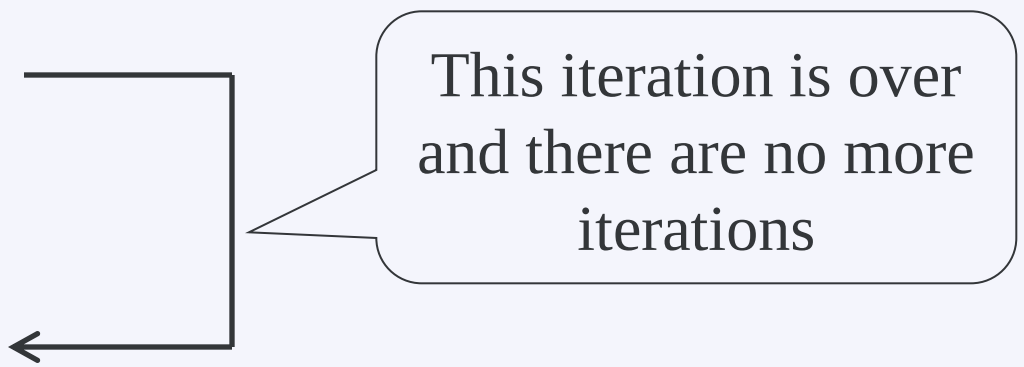


break and continue

- Interrupt for, while or until loop
- The `break` statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The `continue` statement
  - transfer control to the done statement
  - skip the test statements for the current iteration
  - continues execution of the loop

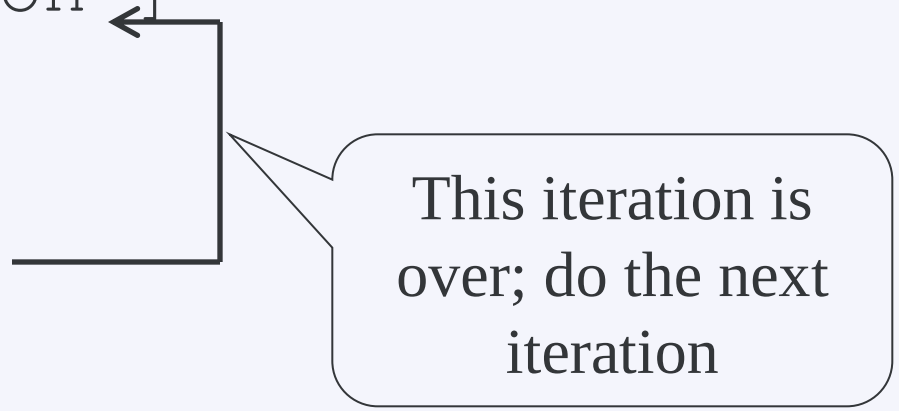
The break command

```
while [ condition ]
do
    cmd-1
    break
    cmd-n
done
echo "done"
```



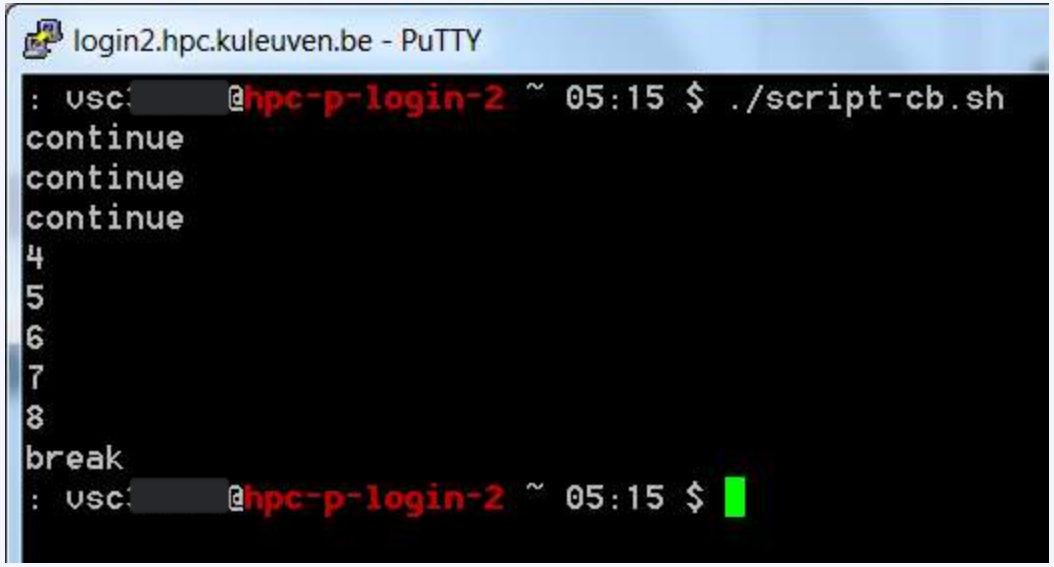
The continue command

```
while [ condition ]
do
    cmd-1
    continue
    cmd-n
done
echo "done"
```

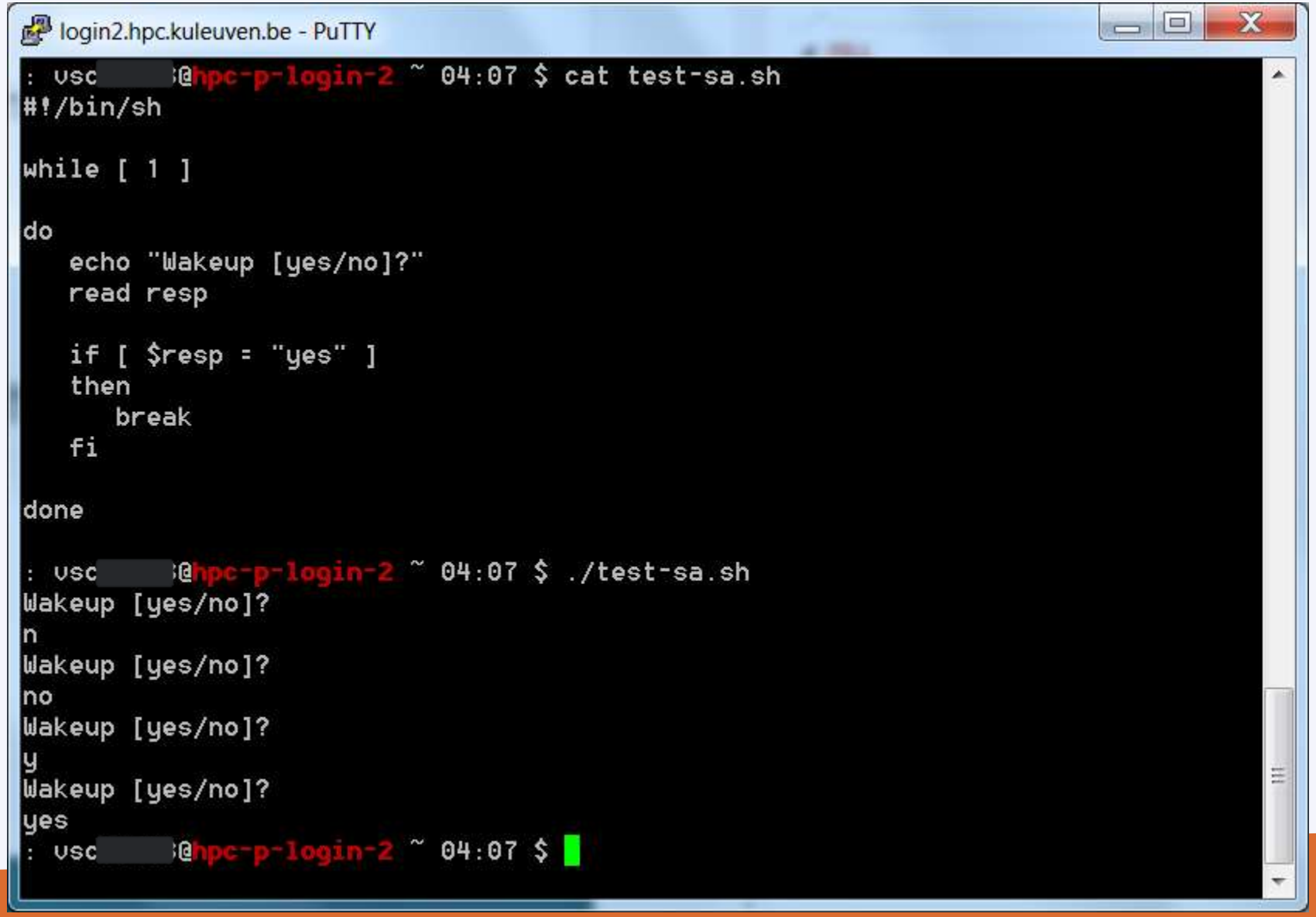


Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
    if [ $index -le 3 ]; then
        echo "continue"
        continue
    fi
    echo $index
    if  [ $index -ge 8 ]; then
        echo "break"
        break
    fi
done
```



break


login2.hpc.kuleuven.be - PuTTY
: vsc[redacted]@hpc-p-login-2 ~ 04:07 $ cat test-sa.sh
#!/bin/sh
while [1]
do
echo "Wakeup [yes/no]?"
read resp
if [$resp = "yes"]
then
break
fi
done
: vsc[redacted]@hpc-p-login-2 ~ 04:07 $ ./test-sa.sh
Wakeup [yes/no]?
n
Wakeup [yes/no]?
no
Wakeup [yes/no]?
y
Wakeup [yes/no]?
yes
: vsc[redacted]@hpc-p-login-2 ~ 04:07 $

continue

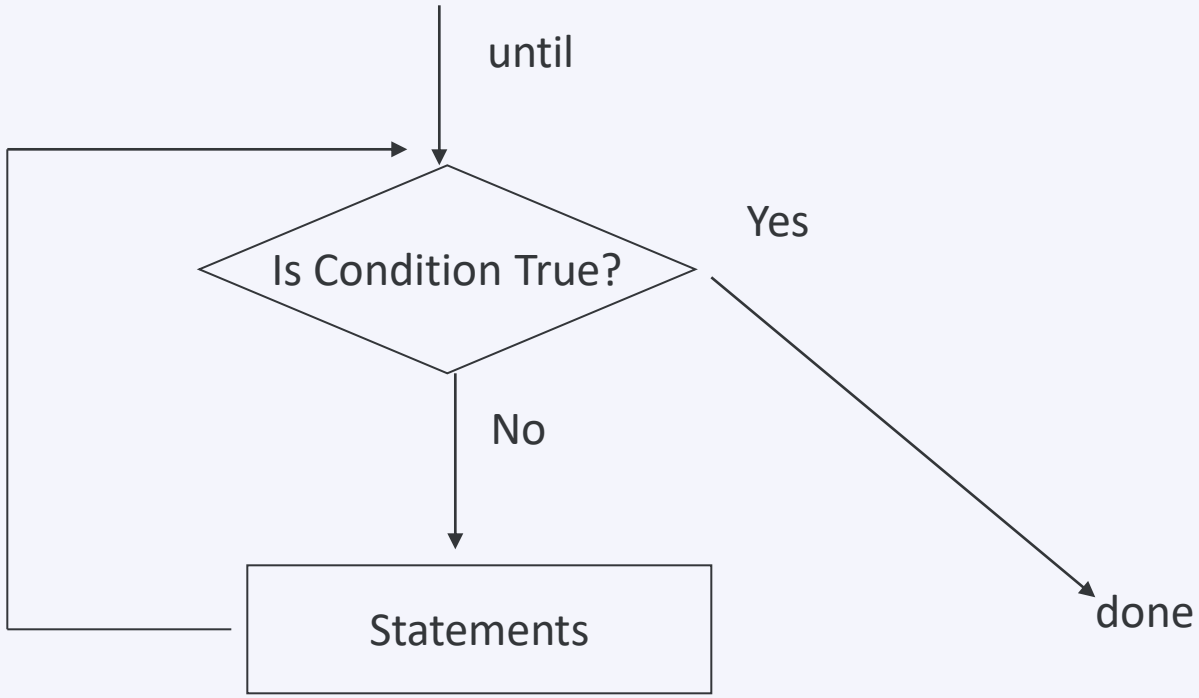
```
#!/bin/bash
for i in 1 2 3 4 5 6 do
### just skip printing $i; if it is 3 or 6
   if [ $i -eq 3 -o $i -eq 6 ]
      then
      continue
### resumes iteration of an enclosing for loop
   fi

   echo "$i"
done
```

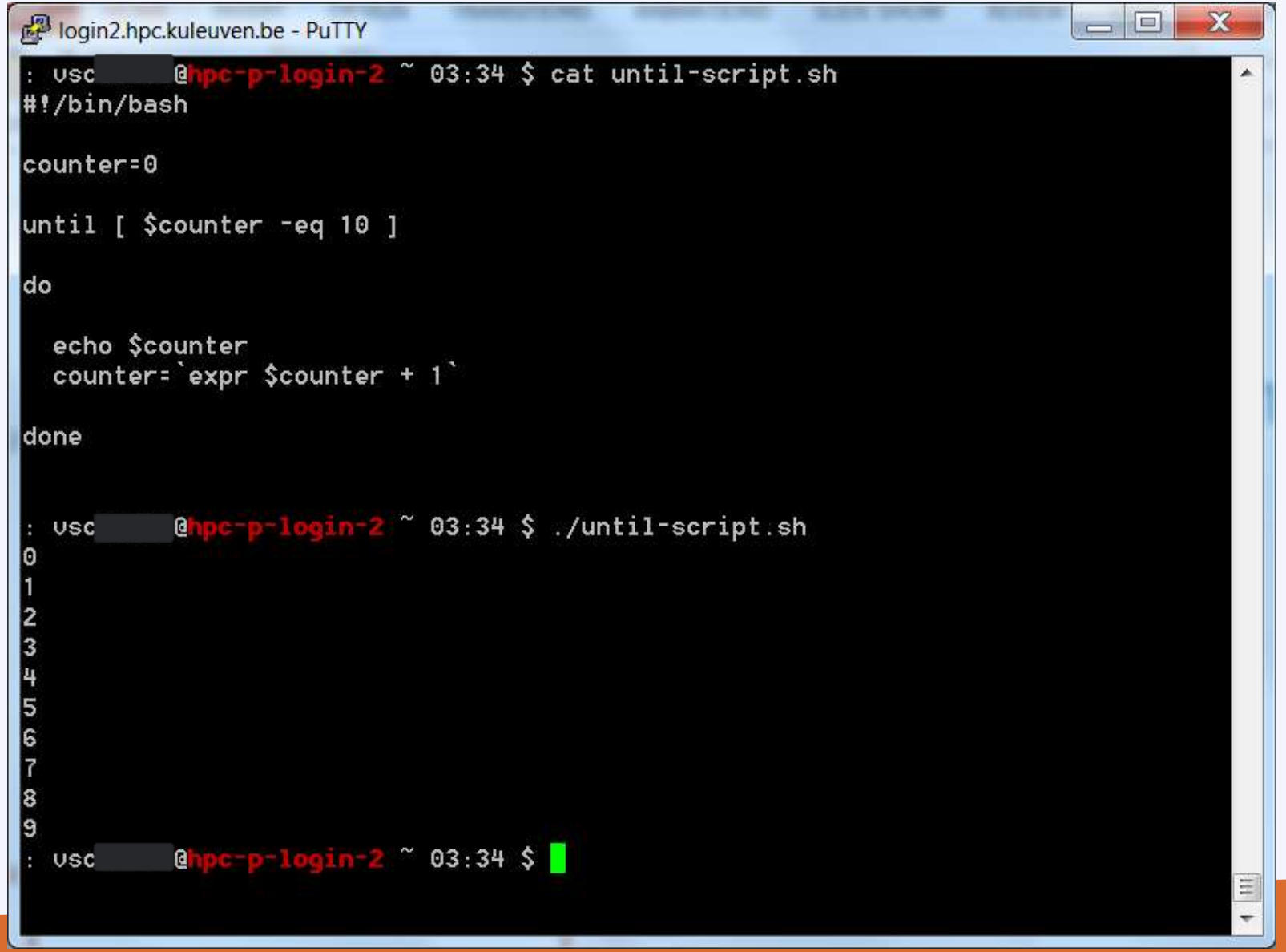
Until loop

```
until CONDITION
do
      STATEMENTS
done
```

Flowchart of until loop



Example of until loop



While vs. Until

login1.hpc.kuleuven.be - PuTTY

```
: vsc[redacted]@hpc-p-login-1 ~ 10:17 $ cat while-script1.sh
#!/bin/bash

counter=0

while [ $counter -lt 10 ]

do

  echo first $counter
  counter=`expr $counter + 1`
  echo again $counter

done


: vsc[redacted]@hpc-p-login-1 ~ 10:17 $ ./while-script1.sh
first 0
again 1
first 1
again 2
first 2
again 3
first 3
again 4
first 4
again 5
first 5
again 6
first 6
again 7
first 7
again 8
first 8
again 9
first 9
again 10
: vsc3[redacted]hpc-p-login-1 ~ 10:17 $
```

login1.hpc.kuleuven.be - PuTTY

```
: vsc[redacted]@hpc-p-login-1 ~ 10:18 $ cat until-script1.sh
#!/bin/bash

counter=0

until [ $counter -eq 10 ]

do

  echo before $counter
  counter=`expr $counter + 1`
  echo after $counter

done


: vsc[redacted]@hpc-p-login-1 ~ 10:18 $ ./until-script1.sh
before 0
after 1
before 1
after 2
before 2
after 3
before 3
after 4
before 4
after 5
before 5
after 6
before 6
after 7
before 7
after 8
before 8
after 9
before 9
after 10
: vsc[redacted]@hpc-p-login-1 ~ 10:19 $
```

Bash shell programming

- Input
  - prompting user
  - command line arguments
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions

Bash Arrays

- An array is a variable containing multiple values. Any variable may be used as an array.
- There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously.
- Arrays are zero-based: the first element is indexed with the number 0.
- Indirect declaration is done using the following syntax to declare a variable:

```
ARRAY[INDEXNR]=value
```

Bash Arrays

- Bash arrays have numbered indexes only, but they are sparse, so you don't have to define all the indexes.

- An entire array can be assigned by enclosing the array items in parenthesis:

```
arr=(Hello World)
```

- Individual items can be assigned with the familiar array syntax (unless you're used to Basic or Fortran):

```
arr[0]=Hello arr[1]=World
```

- But it gets a bit ugly when you want to refer to an array item:

```
echo ${arr[0]} ${arr[1]}
```

Bash Arrays

In addition, the following funky constructions are available:

```
${arr[*]}    # All of the items in the array
${!arr[*]}   # All of the indexes in the array
${#arr[*]}   # Number of items in the array
${#arr[0]}   # Length of item zero
```

Note that the "`@`" sign can be used instead of the "`*`" in constructs such as `${arr[*]}`. The result is the same except when expanding to the items of the array within a quoted string:
`${arr[*]}` returns all the items as a single word, whereas `${arr[@]}` returns each item as a separate word.

Bash Arrays

```
#!/bin/bash
array=(one two three four [5]=five)
echo "Array size: ${#array[*]}"
echo "Array items:"
for item in ${array[*]}
do
  printf " %s\n" $item
done
echo  "Array indexes:"
for index in ${!array[*]}
do
 printf " %d\n" $index
done
echo "Array items and indexes:"
for index in ${!array[*]}
do
 printf "%4d: %s\n" $index ${array[$index]}
done
```

Output:

```
Array size: 5
Array items:
 one
 two
 three
 four
 five
Array indexes:
 0
 1
 2
 3
 5
Array items and indexes:
  0: one
  1: two
  2: three
  3: four
  5: five
```

Bash Arrays

```
#!/bin/bash
array=("first item" "second item" "third" "item")
echo "Number of items in original array: ${#array[*]}"
for ix in ${!array[*]}
do
 printf " %s\n" "${array[$ix]}"
done
 arr=(${array[*]})
 echo "After unquoted expansion: ${#arr[*]}"
for ix in ${!arr[*]}
do
 printf " %s\n" "${arr[$ix]}"
done
 arr=("${array[*]}")
 echo "After * quoted expansion: ${#arr[*]}"
for ix in ${!arr[*]}
do
 printf " %s\n" "${arr[$ix]}"
done
 arr=("${array[@]}")
 echo "After @ quoted expansion: ${#arr[*]}"
for ix in ${!arr[*]}
do
 printf " %s\n" "${arr[$ix]}"
done
```

Output:

```
Number of items in original array: 4
 first item
 second item
 third
 item
After unquoted expansion: 6
 first
 item
 second
 item
 third
 item
After * quoted expansion: 1
 first item second item third item
After @ quoted expansion: 4
 first item
 second item
 third
 item
```

Hands-on 4

1. Write a script calculating average of given integer numbers on command line arguments. Use `for` loop. You can add if statement that checks if at least 2 parameters are given.

2. Write a script to reverse a given integer number (numbers in reverse order). Use `while` block.

3. Write a script that calculates factorial of a given number. Use `until` block.

4. Write a script that sorts the given five numbers in ascending order (using bubble sort algorithm*). Use `for` loop and `array`.

* see https://en.wikipedia.org/wiki/Bubble_sort

http://www.geeksforgeeks.org/bubble-sort/

Shell Functions

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell stores functions in memory
  - shell executes a shell function in the same shell that called it
- Where to define
  - In .profile (.bash_profile)
  - In your script
  - Or on the command line
- Remove a function
  - Use unset built-in

Shell Functions

- must be defined before they can be referenced
- usually placed at the beginning of the script

Syntax:

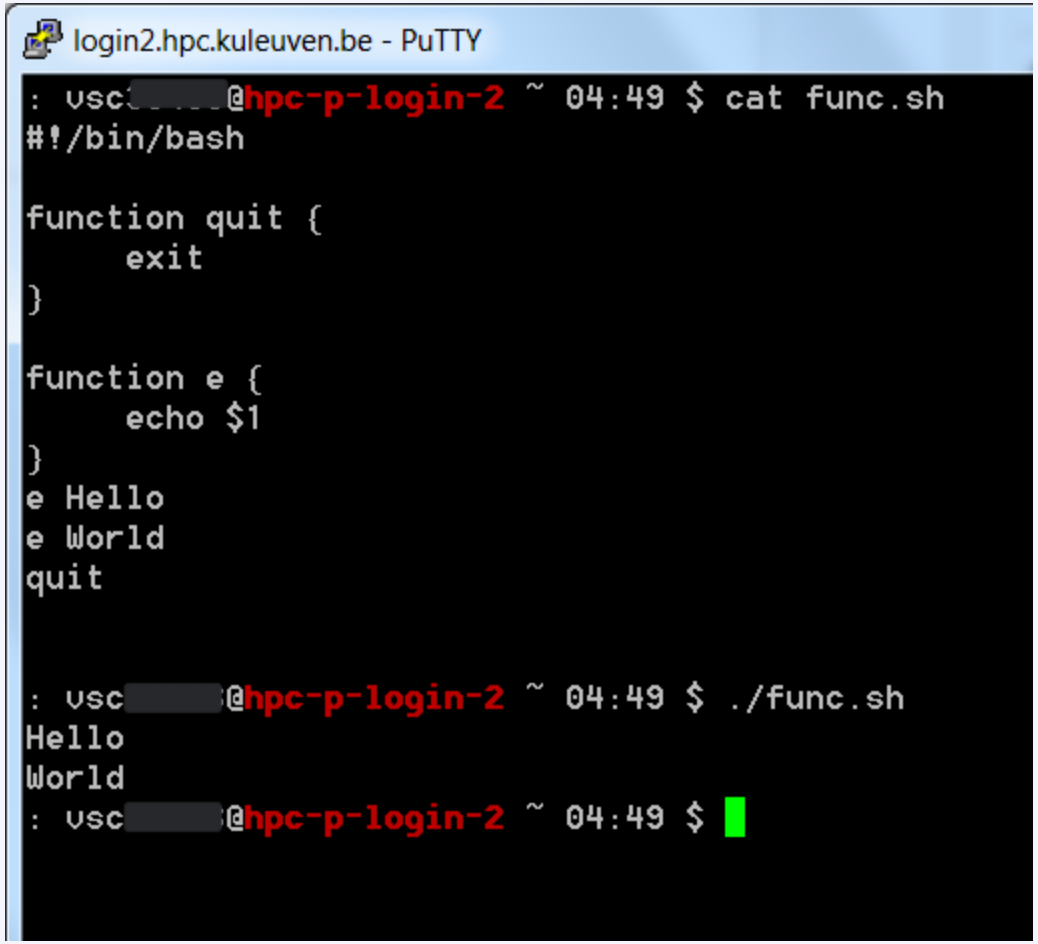
```
function-name () {
   statements
}
```

Example: function

```
#!/bin/bash

function quit {
      exit
}

function e {
      echo $1
}
e Hello
e World
quit
```

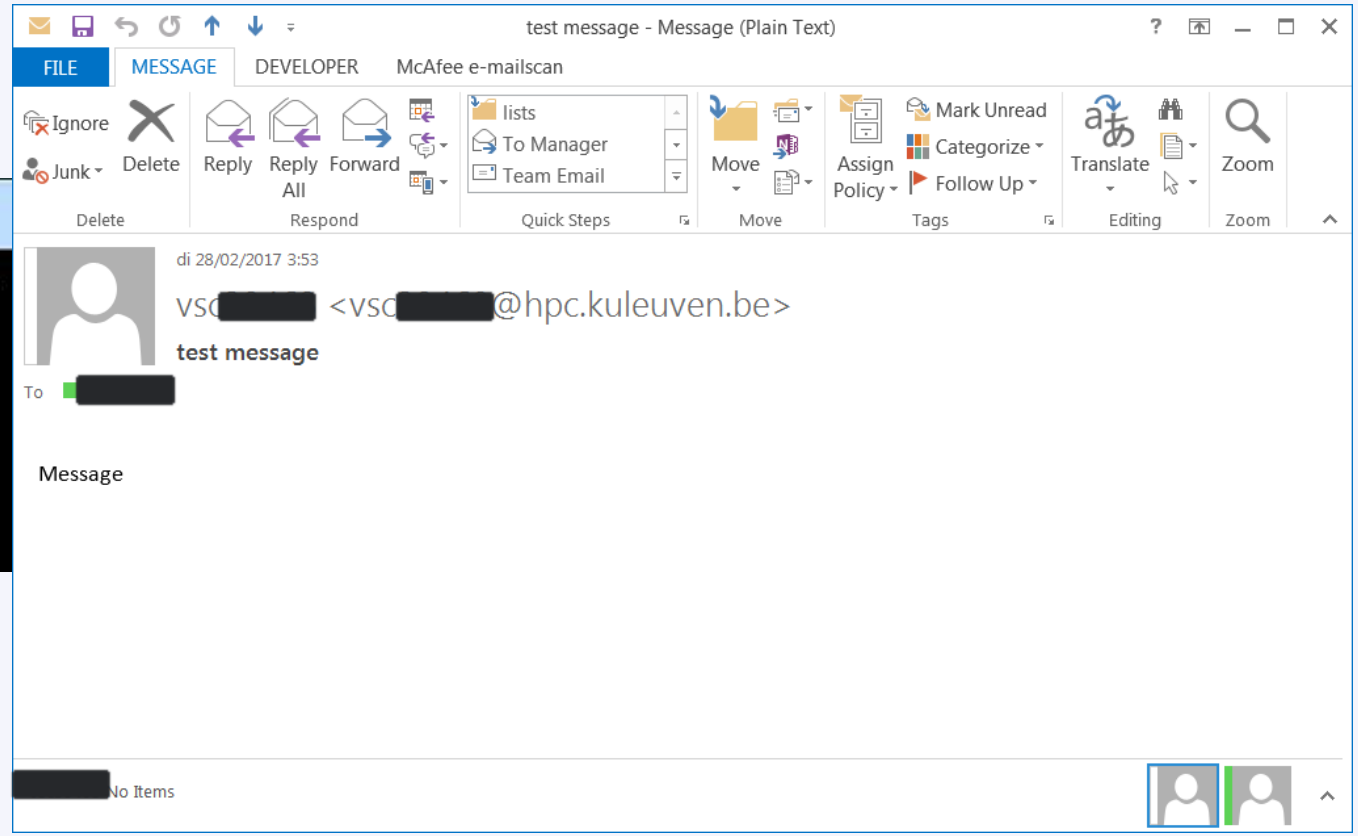


Continuing Lines: \

```
$ echo This \
Is \
   A \
Very \
Long \
 Command Line
This Is A Very Long Command Line
$
```

Email Notification

```
% echo “Message” | \
mail –s “Here’s your message” \
  my.name@kuleuven.be
```



Dates

```
$ DATESTRING=`date +%Y%m%d`
$ echo $DATESTRING
20170227
$ man date
```

Hands-on 5

1. Write a more complicated greeting shell script that based on time (taken from `date`) adjusts the greeting to "*Good morning/afternoon/evening*", etc. in `if` statement.
2. Modify the script so that each if statement is a separate function.
3. Submit the job to the cluster (or just execute it from the login node). The job should:
   a. Copy the file /apps/leuven/training/HPC_intro/helloworldmpi.c into your home directory
   b. Load the intel module
   c. Compile the code: `mpicc helloworldmpi.c -o hello.exe`
   d. Set the value of variable `check` to 2
   e. Increase it by 16
   f. Initialize value of `test` to 0
   g. Set the random number between 1 and 20 as variable `rand`
   h. Check in the until loop how many iterations are needed until `rand` is equal to `check`
   i. Print that information
   j. Check if ~/exercise.txt exists. If so – run the mpi job saving the output to exercise.txt (`mpirun -np $check ./hello.exe > exercise.txt`). If not – first create the file and then run mpi code.
   k. Read the exercise.txt file line by line, assign each line intro array `arr` and print the 5th item of `arr`.

Questions

Helpdesk:
hpcinfo@kuleuven.be or https://admin.kuleuven.be/icts/HPCinfo_form/HPC-info-formulier

VSC web site:
http://www.vscentrum.be/

VSC documentation: https://docs.vscentrum.be

VSC agenda: training sessions, events

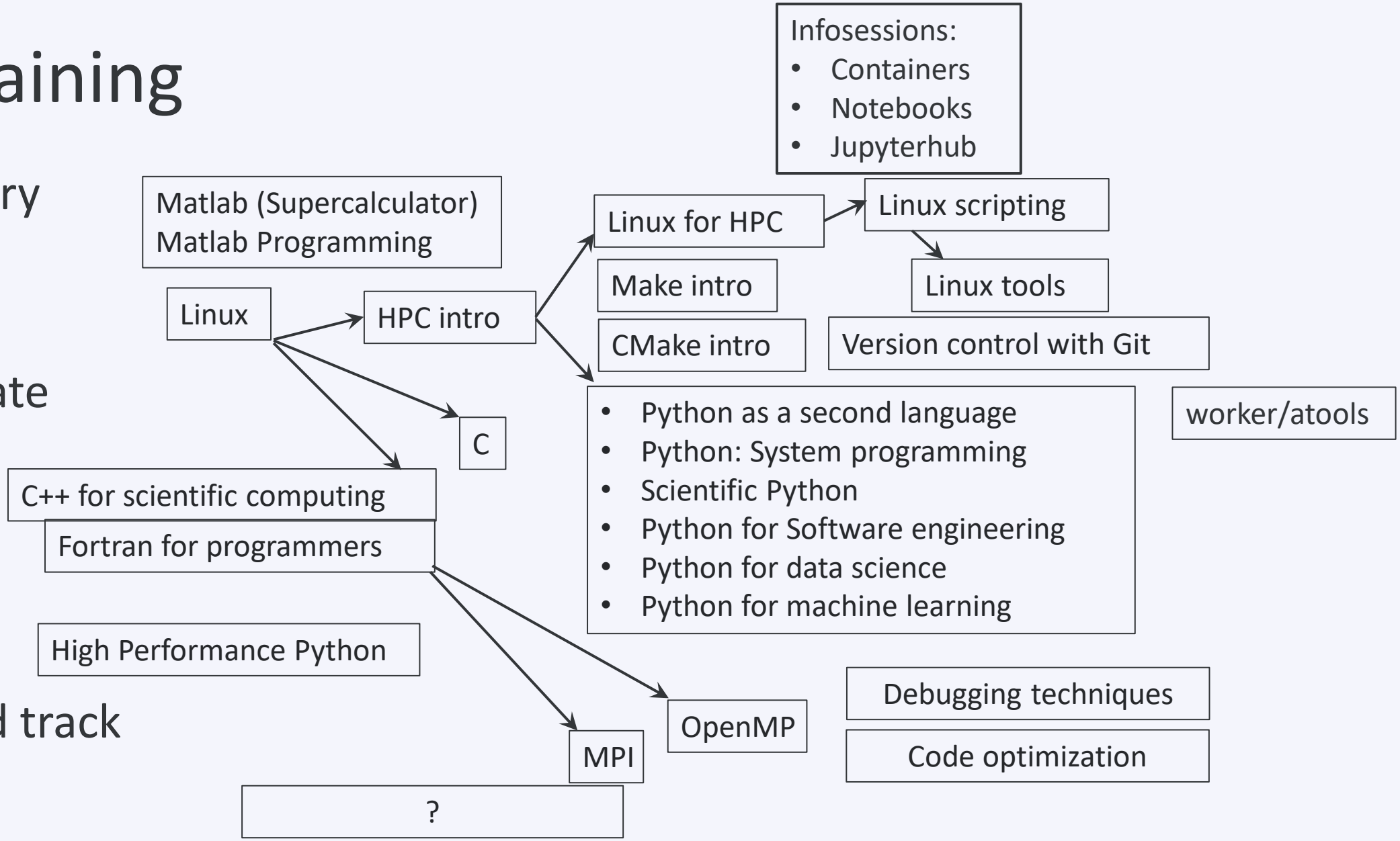
Systems status page:
http://status.vscentrum.be

Stay Connected to VSC

LinkedIn®

VSC training

- Introductory
- Intermediate
- Advanced
- Specialized track



PRACE MOOC Defensive programming and debugging and Fortran for Programmers (announced by e-mail)