BASH and command line utilities
Variables
Conditional Commands
Loop Commands
BASH scripts

SCOMRED, October 2018

The user's initial program – the shell

When a user logins to a system through a terminal, a program is automatically executed, this is an attribute of the user account. If not defined in the user account a system's default initial program will be used.

This program to execute upon a successful login is usually a commands interpreter called Shell, the shell is a command line environment for interaction with the system through typed commands.

The initial program execution is the user session, meaning when this program exits the session ends (the user is logged out). If the initial program execution fails, the user is immediately logged out.

In Linux systems, the **/etc/shells** files contains a list of allowed shells on the system, if a user's initial program is not there he will not be able to login. In Linux the default Shell, of not defined otherwise in the user account, is /bin/bash, the BASH (Bourne-again shell).

The shell

When a user logins to a system through a terminal, the shell is executed, in a Linux system this means a process is created, so each used logged in will have a running shell program dedicated to him.

Every shell establishes a command line environment, with several status data, some of them are:

- The current working directory (CWD) - when a filename is referred in a relative way (not starting by /) this is the reference starting point.
- The shell prompt - notifies the shell is ready for a new command and also presents useful information, often the username, the hostname and the current working directory. Also the # character in the prompt usually denotes the user is a super user.
- Environment variables – they hold values that represent the configurations to be used and current status.

In Linux each process has several data attached to it, and that also applies to the shell process. They encompass the current working directory and environment variables for several purposes. For instance the environment variable named **PS1** is used to store the prompt format to be displayed by the shell to the user.

BASH (Bourne-again shell)

BASH (/bin/bash) stands for an improved version of the Bourn Shell (/bin/sh). It's the standard shell for most Linux operating systems and is widely used in many other devices.

Every shell is able to understand a set of commands by itself known as internal commands, if an unknown internal command is typed, then the shell assumes it must be an external command.

External commands are not implemented by the shell itself they must be available on the filesystem in the form of an executable file (a file with the x permission), if so the shell creates a new process to execute the file (transform it into a running process).

The list of folders where executable files are searched sequentially is stored in the environment variable named **PATH.** If a typed command is not internal and an executable file with that name is not found in any of the folders defined in the PATH, then the shell will simply tell **command not found.**

However, even if the executable file is nowhere in the PATH folders, it may still be executed if the full path to it is used.

Environment variables

Of course internal commands trend to run faster because it's not required to create a separate process an load into it the executable file. Thus, for the sake efficiency, the BASH implements several internal commands that mimic external commands.

The external command **which** searches for a given command in the PATH and shows where the corresponding executable file is, for instance **which echo** will shown the **echo** external command is **/bin/echo**, however, if you type the **echo** command in the BASH, an internal version is used instead, to use the external version you must directly type **/bin/echo**.

Variables are referred by names and they are places to store data, some variable names are used for specific purposes (e.g. PATH and PS1), unless we want to act on that specific purpose those names should be avoided.

Beyond this notice, variable names may by arbitrarily defined at will. Data stored in a variable is as well arbitrary, they are simply strings of characters.

Variables – assignment and reference

The assignment operation is implemented trough the equal signal by typing the command:

NAME=VALUE

Where NAME is the variable name and VALUE the value to be assigned to it. If the variable already existed, the previously stored value is replaced, and thus lost, otherwise a new variable name is defined.

Blank characters like space and tab are used by the shell as separators of commands and elements in a list, thus to assign to a variable a value that contains those characters the value should be placed between double quotes.

NAME="NEW VALUE"

The value stored in a variable may be fetched at any time by referring the variable name with the $ prefix, for instance **echo $NAME** prints on the terminal the content of the variable named NAME. If variable named NAME is not defined no error will occur, any undefined variable is valid, it's simply regarded has having an empty content.

Variables – expressions processing

To remove a variable, or undefine it, we simply assign to it an empty value:

```
NAME=
```

The BASH processes (expands) expressions that contain special characters it recognises, this happens on every command, but is very useful in assignment commands, if the value contains those special characters it's expanded and the result is then assigned to the variable.

Let's see some relevant cases of expressions the BASH will process.

$(command) – executes the command.

For instance the command **LSRESULT=$(ls -la /)**, assigns to variable named LSRESULT the result (output) of the **ls -la /** command.

${VARNAME} – evaluates to the value of variable named VARNAME.

The curly brackets are optional, they may be required to clearly identify the variable's name, for instance when concatenating a string to a variable, for instance:

```
A=${A}TEST
```

Will store in the A variable its previous value with TEST appended.

Expressions processing

$((arithmetic expression)) – evaluates arithmetic expression.

For instance the command SUM=$((200+300)), assigns to variable named SUM the value 500. Of course instead of constants we could also use variables, for instance:

A=200; B=300; SUM=$(($A+$B)); echo $SUM

Will echo the value 500 on the terminal. When evaluating an arithmetic expression we must be sure used variables contain numbers not strings otherwise an error will occur, also arithmetic expressions are limited to integer numbers floating-point numbers are not supported.

To use floating-point numbers an external program like **bc** could be used.

From the example above, we can also see on a single command line, a semicolon separated sequence of different commands can be executed, they are executed one at a time, one after the other from left to right.

Processes in Unix

The use of a shell in a Unix system, like BASH in Linux, can be better understood if we know how Unix Operating Systems in general and Linux in particular creates processes and executes commands.

When the Linux kernel boots, it will start with one single process. The name of the executable file to be loaded into this initial process is passed to kernel through the **init=** argument, usually it's **/sbin/init**. All and every process running in the operating system comes up from this initial process.

In Unix operating systems, a new process is created by calling the kernel provided system-call **fork()**, but it might as well be named clone. When a process calls fork() the kernel creates an exact copy of the calling process, it's really an exact copy, everything is equal, code being executed, data and environment, it also includes open files, if a file is opened (in use) in the original process it will also be opened in the new process.

In fact, there is only one difference between the original process and the copy, each process is identified by a unique number called PID (Process Identifier), and that is the only difference. The original process, also called parent, keeps the same PID it had, the new process, also called child, has a new unique PID assigned to it.

Commands execution in Unix

To start a new process we already know the kernel's system-call fork() is used, it creates an exact clone of the current process (parent), called child.

When a child process terminates it returns to the parent process an exit status code, it's an unsigned integer number. By convention the exit code **zero stands for a successful completion** of the child process (mission accomplished), an exit code greater than zero means some error has occurred, the value itself may be used to represent the error. The exit code is settled by the child process when it exits, and is then passed by the kernel to the parent process.

The kernel provides another basic system-called called exec(), what exec() does is replacing the code being run in the current process with the code loaded from an executable file and executes it from the start. This is how an application stored in a file is transformed into a running process.

So, making it simple, to run a command we first use fork() and then exec() on the child process.

The exec() function replaces only the code being executed, all other characteristics of the process are thus inherited by the new running code, including PID, environment variables and opened files.

Processes interaction

Processes in Unix/Linux are closed boxes, meaning each process can use only what is within the process. This makes the operating system very stable.

Interactions of a process with the outside world (other processes and system resources) are strictly controlled by the kernel.

Some relevant interactions with the outside world are:

- **Exit code** - when o process ends it settles an exit code, that is transmitted to its parent process only.
- **Signals** – it's possible to send a signal to a process, as far as the sender has the appropriate permissions. The sender process must be owned by an administrator or by the same user the target process belongs to. When a process receives a signal it interrupts what it was doing and executes a function defined for that signal (signal handler), then returns back to what it was doing.
- **File descriptors** – to access external resources, a process must request that to the kernel (open a resource), as result the kernel assigns to the process a file descriptor that represents the requested resource and allows access to it.

File descriptors in Unix

In Unix/Linux file descriptors are unsigned integer numbers, usually every process has three opened file descriptors that are inherited from its parent (admitting they were opened in the parent process), they are:

0 – named **stdin**, usually assigned to a keyboard device, allows a read only access.

1 – named **stdout**, usually assigned to a display device, allows a write only access.

2 – named **stderr**, usually also assigned to a display device, again allows a write only access, but is meant to be used when presenting error messages, unlike stdout that is meant to display normal messages.

The term **file descriptor** is misleading, in fact they are used to handle files, but they are also used to access a wide variety of other type of resources, including for instance network connections (sockets) and IPC (Inter Process Communication) resources like pipes, shared memory and others.

Now we have some background knowledge about Unix/Linux operating systems, we will process with the BASH exploration.

BASH – conditional commands

Conditional commands are commands the include a decision, they are all internal commands of the shell, mainly because they are compound commands. Decisions are made depending on something being true or false, usually that would be a logical expression that can be evaluated as true or false.

However, in the BASH true or false are the exit code of a command, if a command exits with code zero (success) than that is true, if it exits with other value, for instance one, that stands for false.

There are two dumb commands that can be used for this purpose, **true** command (always exits with code zero) and the **false** command (always exits with code one).

One very simple BASH conditional command is **if**:

if {commands1;} then {commands2;} [else {commands3;}] fi

It executes {commands1} if their exit code is zero, then it will execute {commands2}, otherwise, if else is used, then it executes {commands3}.

The mentioned commands, can be a single command or a sequence of semicolon separated commands. Regarding {commands1} the exit code that maters is the exit code of the last command in the sequence.

BASH – if command examples

Try figuring out what these command lines will display:

```
if true; then echo OK; else echo KO; fi
```

```
A="THE RESULT IS: "; if true; false; then A=${A}TRUE; else A=${A}FALSE; fi; echo ${A}
```

```
if (ls /; exit 0) ; then echo "TRUE"; else echo "FALSE"; fi
```

In the last case commands were placed within brackets, this has a special meaning for BASH. It means execute it in a separated child process, if not done this way what would happen is the exit command would exit the BASH being used itself.

Any command's exit code can be used, but there's one particularly useful command then can be used here, it's the **test** command:

test EXPRESSION

Where EXPRESSION is a logical expression to be evaluated as true or false, and thus the **test** command exits with the corresponding code.

The test command supports several logical operators like equal an not equal, greater than and so on.

BASH – the test command

The test command supports a wide variety of logical operators, as usual you can take knowledge of them by going to the corresponding manual page (**man test** for the external version or **man bash** for the internal version, they should work the same way).

Examples:

```
A=10; if test $A -gt 20; then echo GREATER; fi
```

```
A=30; if test $A -gt 20; then echo GREATER; fi
```

The -gt operator stands for grater than, it only works for numbers. You really should take some minutes to read the manual and take a look on supported operators, multiple conditions can also be chained by connecting then with the AND (-a) or OR (-o) operators.

The test command may be also used in the form [EXPRESSION], so above examples can be also implemented by:

```
A=10; if [ $A -gt 20 ]; then echo GREATER; fi
```

```
A=30; if [ $A -gt 20 ]; then echo GREATER; fi
```

There is yet another way to implement the AND/OR/NOT operations without using the test command that may be useful.

BASH – exit codes and logical operators

When executed a command returns an exit code than is interpreted as TRUE (equal to zero) or FALSE (not equal to zero).

For a sequence of commands the exit code of that sequence depends on how commands are putted together:

command1; command2 - as mentioned before, the exit code is the one returned by command2 (the last command in the sequence).

command1 && command2 – the exit codes are combined by the AND logical operator, so it will result in zero (true) only if both result in zero (true). One interesting feature is that if command1 returns false (not equal to zero), **then command2 is never executed** because that's not required to evaluate the overall result.

command1 || command2 - the exit codes are combined by the OR logical operator, so it will result in zero (true) any of the commands returns zero (true). Again, one interesting feature, is that now if command1 result is zero (true), **then command2 is never executed** because that's not required to evaluate the result.

! command1 – the exit code is the logical negation (NOT) of the command1 exit code, if command1 returns zero it's turned into 1, otherwise (false) it's turned into zero (true).

BASH – input and output redirection

When running in a terminal, the BASH is itself a process, to interact with the user, this process has three open file descriptors, stdin (0) for reading, stdout (1) and stderr (2) for writing, they were created by opening the device file (in system folder /dev) that represents the terminal being used, for instance /dev/tty0 or /dev/tty1.

Because child processes, among other things, inherit the parent's file descriptors, commands and applications started from the BASH command line will also have those three file descriptors opened for that same device.

When a command is executed in BASH it's possible to redirect these descriptors so that other files are used instead. The greater than symbol (>)redirects descriptors to a file in write mode, so it should be used with descriptors 1 and 2, the less than symbol (<) redirects descriptors to a file in read mode, so it should be used with descriptor 0.

The general forms of application are: **n<&m** **n>&m**

Where n stands for the command's internal file descriptor and m stands for the external file descriptor of the overall command with redirections applied, in both cases **&m** may also be a filename instead.

BASH – input and output redirection

To be more precise the syntax is as follows:

[n]<&m|filename **[n]>&m|filename**

This means the command's internal descriptor may be omitted. When redirecting for reading (<), then if n is not specified, 0 (stdin) is assumed. When redirecting for writing (>), then if n is not specified, 1 (stdout) is assumed. So **>** is equivalent to **1>** and **<** is equivalent to **0<**.

When redirecting to write into a file, instead of >, >> can be used. In both cases the file is created if it doesn't exist, however with a singe > the file is rewritten if it already existed (the previous content is erased), with a double > the new content is appended to the previous content.

Examples:

```
command > teste.txt
command 2>> errors.txt
command > teste.txt 2>> errors.txt
command 2>&1
command 1> teste.txt 2>&1 < input.txt
command 2> teste.txt 1>&2 < input.txt
command 1>&1 2>&2 0<&0
```

BASH – pipelining commands execution

Often it may be useful to use the output of a command as input to another command. We could do that by using a temporary file to store the output of the first command, and then use it as input for the next command:

command1 > tmp.txt ; command2 < tmp.txt; rm tmp.txt

Yet there's a simpler way to do the same through a pipe:

command1 | command2

If we want command2 to read on its stdin, not only command1's stdout, but also its stderr, the following variation can be used:

command1 |& command2

It's equivalent to: command1 2>&1 | command2

This is called a pipeline of commands, the exit code of a pipeline is the exit code of the last command in the pipeline.

Bear in mind, regarding both redirections and pipelining, options presented here don't include all available features.

Scripts in Linux/Unix

Scripts are text files with instructions/commands to be executed by an interpreter program, so interpreted languages are used. The BASH also works as interpreter, of course, for scripts with BASH commands, they are named BASH scripts.

In order for a script to become a new useable command in a Unix/Linux system, some conditions must be met:

- For users to be able to execute the script, they must have the execute (x) permission on the script file, and they must also have the read (r) permission. The read permission is not required to execute binary (compiled) programs, but it's required to execute script files.
- The script file should be located in some folder that is included in the PATH environment variable, otherwise, to run it from the command line the full path to the file would have to be used.
- It must be recognized by the operating system as being a script, and more precisely a BASH script. Unix/Linux systems identify the file type by it's content and not by the filename like in Windows systems (filename extension). The file content is matched with a list of known magic numbers, a magic number is a set of fixed values bytes that exist is fixed positions of the file. For scripts, the magic number is characters #! in the first two positions.

BASH Scripts

In Unix/Linux, the first line of every script starts with **#!**, following that sequence, in the same line, the interpreter program should be declared. For a BASH script it will be **#!/bin/bash**, that's the first line of every BASH script. When the kernel is asked to execute such a file it will first execute the interpreter program an then pass to it the file content for it to execute.

Because for interpreted languages any line started by # is regarded as a comment by the programmer, it's ignored in runtime, the **#!/bin/bash** line doesn't interfere with the program execution.

Each line in a BASH script contains a command exactly as we would type it at the command line, so each line can also be a list of commands putted together with characters ; && !! |

The most important concept is, commands in each line are executed on after the other, one at a time, starting with the first line until the last line is reached.

When the last line is reached, the script ends with exit code 0.

At any point in the script the **exit CODE** command can be used to exit with code **CODE**, beware exit is a BASH internal command if called from a script it exits the BASH that is running the script, so exits the script, if called from the BASH command line it exits the login BASH and consequently logs out the user.

BASH Scripts – command line arguments

Actually, a BASH script is always equivalent to a single very long command line with the whole sequence of commands in the script. Commands in successive lines within the script are equivalent to commands separated by semicolon in the command line.

But there are several advantages on using BASH scripts. For a complex and long commands sequence in a single line, it's humanly unbearable to analyse and edit it, specially if compound commands are used.

The script is a file, so it's persistent and by placing it in an appropriate folder, with appropriate permissions, it becomes a new command available to all users.

Common software development techniques can be used when developing a script, like for instance incremental programming.

Command line arguments used when starting the script are available through variables:

$0 – the command name itself (script), argument zero.

$1, **$2**, … – the first argument, the second argument, and so on.

$# - the number of arguments (excluding argument zero).

$* - a list with all arguments (excluding argument zero).

The **shift** command may be used to shift left values $1 $2 $3 …

BASH Scripts – compound commands

One thing scripts turn much more clear to the programmer is compound commands, we have already mentioned the if command:

if commands1; **then** commands2; **[else** commands3;**]** **fi**

It's called compound because it has several parts (if/then/else/fi), with commands within those parts. Soon we will see more compound commands. When used in a script, compound commands become more clear if split into several lines. After any part of a compound command, changing to a new line is ok, of course after every commands sequence the semicolon may be replaced by changing to a new line as well.

So, the if command may be used any of these ways in a script:

```
#!/bin/bash
if
commands1
then
commands2
else
commands3
fi
```

```
#!/bin/bash
if commands1; then
commands2
else
commands3
fi
```

```
#!/bin/bash
if commands1; then commands2
else commands3
fi
```

BASH Scripts – loops

Loops allow the repeated execution on the same commands, loops are also compound commands.

while commands1; do commands2; done

This command executes commands1, if they return true (0), then executes commands2, this repeats until commands1 return false, in that case commands2 are not executed and the loop ends (done).

until commands1; do commands2; done

Same as before, but the loop keeps running until commands1 return true and not while they return true.

From within the loop execution, an immediate end of the loop may be forced by calling the internal command **break**.

When implementing loops, programmers must be cautious and ensure the condition established to stop the loop is ever going to happens, otherwise it will be an infinite loop and the program will never end.

BASH Scripts – for loops

The **for** internal command has two alternative usages for creating a loop, in both cases break may be used to end it:

for VARNAME in list; do commands1; done

The variable **VARNAME** is assigned sequentially each value in **list**, and for each, commands1 are executed. Of course, in each commands1 execution the variable's value is available ($VARNAME).

for ((expr1; expr2; expr3)); do commands1; done

This second usage is for arithmetic operations only, expr1, expr2, and expr3 must be arithmetic expressions resulting in an integer. If omitted these expressions will be accounted as having result 1. The arithmetic evaluation by BASH has its own rules, one is, within these expressions variable names may be referred without the $ prefix.

This **for** command does the following. First it evaluates expr1, then it evaluates expr2, if not zero, then executes commands1, then evaluates expr3, and finally evaluates expr2 again. Now the loop either continues, if expr2 is not zero, or stops.

Example: **for ((i=1000;i;i--)); do echo $i; done**

This example prints numbers starting with 1000 down to 1.

BASH Scripts – case

The internal **case** command matches a value with a sequence of patterns, for the first match found, the corresponding commands are executed.

case VALUE in pattern1) commands1;; pattern2) commands2;; esac

If commands end with a double semicolon as presented above, no further matches are tried after executing commands. If commands end with **;&**, then commands in the next pattern are also executed. If commands end with **;;&**, then one additional matching is tried against the next pattern.

Patterns use special characters for matching:

***** - matches any sequence of characters, including an empty one.

? – matches any single character, not an inexistent character.

[chars**]** – matches one of the enclosed characters. If chars is made of two characters separated by an hyphen, that's a range (e.g. [A-G]). The sense may be reversed by using **[!**chars**]**, it matches any character except those specified. Character classes may be used in the form [:classname:], some valid class names are **digit**, **upper**, and **lower**.

For instance, pattern **[:upper:]??[:lower]** matches strings with 4 characters, starting with an up case letter and ending with a low case letter.

BASH – variables expansion

In every command line, before executing it, the BASH expands some sequences started by special characters. This is the case of variables, $VARNAME or ${VARNAME} are expanded to the variable named VARNAME current value. While expanding a variable, some interesting manipulations are possible, there's a huge number of options, they may be found under section **Parameter Expansion** of the BASH manual (man bash). Braces are now mandatory, some examples:

${#VARNAME} – expands to the VARNAME value's length (number of characters).

${VARNAME:offset:length} – expands to a substring of VARNAME's value, starting at position number offset (the first position has offset value zero). The form ${VARNAME:offset} assumes length is up to the end of the string.

${VARNAME#pattern} – prefix pattern matching. If pattern matches a left part of the value, then the matching part is removed from the expansion. With a single # the smallest matching part is removed, with a double hash (##) the longest matching part is removed.

${VARNAME%pattern} – suffix pattern matching. If pattern matches a right part of the value, then the matching part is removed from the expansion. With a single % the smallest matching part is removed, with a double percent (%%) the longest matching part is removed.

BASH Script – functions

Defining a function in a script has similar effects to creating a new external command in a separate script.

A function has a name and a set of commands to be executed, it may be declared as:

function FUNCTION-NAME() { commands; }

If parenthesis are used the keyword function is optional, if the keyword function is used then parenthesis are optional.

After being defined (not before) the function may be called as with any other command:

FUNCTION-NAME arg1 arg2 arg2 …

Within the function, variables $#, $*, $1, $2, … represent arguments provided when calling it, however, $0 keeps the value it had. The function's exit code is it's last executed command's exit code.

Unlike with executing an external command or script, when a function is called, the caller BASH process is used and no new BASH process is created to run the function, this has some consequences.

BASH Script – functions and source

Calling the **exit** command from within a function ends the calling command, this is because this command exits the shell and both the calling command and the function share the same shell process. Instead, the **return** internal command should be used to leave a function and go back to the command after the calling point.

Sharing the same BASH process means variables are also shared, so variables available on the calling point are also available in the function, and vice versa.

A BASH script may be split into several files, by using the **source** command, a text file with BASH commands may be included in a script:

source FILENAME

One way to describe it is, this command is replaced by the content of file FILENAME, the file isn´t required to be a script or have the execute permission. If FILENAME doesn´t contain a slash, then it will be searched in PATH.

Functions are supposed to be reusable, so one scenario would be having those functions definitions in a shared file to be included by several scripts through the **source** command.

External command line utilities

One thing a programmer should do is taking the most of what already exists and never write what has already been written, unless there's a good reason for that.

A vast number of external commands are available in Unix/Linux, knowing them is going to save a lot of work when developing scripts in BASH.

Many of these commands are dedicated to text contents processing, by default they receive text in stdin, processes it and print the result in stdout, some often used commands are:

cat – Does nothing, outputs its input. The **tac** command reverses lines' order (last line first).

tr – Outputs lines with some translated or deleted characters as defined by arguments.

cut – Outputs parts of the lines, for instance characters in fixed positions of each line, or fields, by using a field delimiter.

grep – Outputs only lines that match, or don't match, a pattern.

sort – Outputs a sorted content, following specifications given by arguments.

head and **tail** – Outputs the first or last lines.

Other useful commands

date – handling hour and date, a wide variety of options and formats are supported.

find – finds objects in the filesystem, according to specified properties. Outputs object's properties as requested.

printf – prints data using a similar format to the printf function in C language. Allows more precise output formatting than the echo command.

read – reads a line from stdin an assigns values to variables.

The text editor

Scripts are text files, to create and edit text files a program usually called **text editor** is required, some options available for the command line are:

vi – Standing for Visual Editor. This is the best command line text editor, however, it's not easy for beginners.

nano – User friendly text editor.

mc – The GNU Midnight Commander is a friendly file manager with its own text editor. It includes mouse integration and it may be the best option for beginners.

BASH scripts – example 1

```
#!/bin/bash
echo "This command reads a list of positive integer numbers and calculates the biggest,"
echo "smallest, and the average"
read -p "Enter a positive integer number please (0 to end): " V
MIN=$V; MAX=$V
NUM=0; SUM=0;
while [ $V -gt 0 ]; do
 if [ $V -gt $MAX ]; then MAX=$V; fi
 if [ $V -lt $MIN ]; then MIN=$V; fi
 NUM=$(($NUM+1))
 SUM=$(($SUM+$V))
 read -p "Enter a positive integer number please (0 to end): " V
done
echo "$NUM numbers entered"
echo "MAX = $MAX"
echo "MIN = $MIN"
echo "SUM = $SUM"
if [ ${NUM} -gt 0 ]; then
 echo "Integer average (truncated) = $(($SUM/$NUM))"
fi
```

When receiving integer numbers from the user, entered data should be validated, although we are expecting a number it can be anything.

On the next example the readAndValidateV() function reads the user input and checks if it has only numeric digits. This is just one, possibly not the best, solution to overcome this issue.

BASH scripts – example 2

```
#!/bin/bash
echo "This command reads a list of positive integer numbers and calculates the biggest,"
echo "smallest, and the average"
function readAndValidateV() {
 read -p "Enter a positive integer number please (0 to end): " V
 HAS_ALPHA="$(echo "$V"|tr -d "[:digit:]")"                    # remove all digits
 if [ -n "$HAS_ALPHA" ]; then                                  # not empty
  echo "Invalid number entered, zero assumed"
  V=0
 fi
}
readAndValidateV
MIN=$V; MAX=$V
NUM=0; SUM=0;
while [ $V -gt 0 ]; do
 if [ $V -gt $MAX ]; then MAX=$V; fi
 if [ $V -lt $MIN ]; then MIN=$V; fi
 NUM=$(($NUM+1))
 SUM=$(($SUM+$V))
 readAndValidateV
done
echo "$NUM numbers entered"
echo "MAX = $MAX"
echo "MIN = $MIN"
echo "SUM = $SUM"
if [ ${NUM} -gt 0 ]; then
 echo "Integer average (truncated) = $(($SUM/$NUM))"
fi
```