Scripting languages, BASH & co

Tomasz Bold
tomasz.bold@fis.agh.edu.pl
D11 – pok. 107

Outline

- The **bash** shell
  - Other shells?
- Shell scripts
  - variables, controls, functions and aliases, redirections, pipes, ...
- Associated programs
  - awk, sed, ...

Shells

- bash – Bourne Again SHell – successor of sh
  - a bit newer than bas bash is the zsh – syntax ~identical
- ksh & kcsh
- csh & tcsh

Hello Word

```
#!/bin/bash

echo Hello World!
```

Internal vs external commands

- Shell (bash) defines several internal instructions
  → we will walk through some of them

- External programs can be invoked in an identical manner
  (no matter what they are: programs in C, scripts in python or bash itself)

Single use script

- The minimal (typical) script can be as short as one line

```
du | sort -n -r | head -5
```

du == disk usage

Send result to the next command

Sort numerically in reverse order

List first 5 lines

Script execution vs loading

- Shell scripts can be made executable programs

```
chmod +x zd.sh
```

- so they can be later executed from another script or command line:
  `./zd.sh`

- The script can also be loaded:
  `source zd.sh`
  - does not need to be executable
  - variables declared in this script are accessible in caller shell … will clarify that in a moment

I/O

3 streams

- Each UNIX process starts with 3 available streams
  - stdout: buffered, default output eg: `echo hello`
  - stderr: unbuffered, useful for errors
  - stdin: standard input
- This streams can be redirected to a file or pipe
  - Everything in UNIX is a file …

Redirecting *out

ls > listing.txt

redirects stdout to a file

ls 2> errors.txt

redirects errors to a file (stdout not redirected /printed on the terminal screen)

ls > all.txt 2>&1

ls &> all.txt

join stdout & stderr and send to a file

@@ Useful file willing to accept any output `/dev/null`

```
ls 2> /dev/null
```

Redirecting stdin

- When the program reads data from the console:

```
number of iterations: 100
step length: 0.01
file name: wyk_100_001.data
```

We can store the „human” input in a file:

```
100
0.01
wyk_100_001.data
```

and execute like this:

```
prog < input > wyniki
```

Input redirection

Piping

- Data produced in one program can be forwarded to another program for further processing
- Done by a command: |
- Example: pic files containing xyz

```
ls | grep xyz
```

one can connect as many programs as needed. This is very efficient as all communication is performed in memory. Final result can be sent to a file with the „>”.

@@ useful program **tee** copies all input to a file and stdout

```
progX | tee log | grep ERROR
```

Shell scripts

Variables

- For a longer scripts it is useful to be able to define variables
- Variables in bash can only be assigned
    - no build-in operations:

```
STR=“Welcome”
```

    - small and CAPITAL letters are distinguished
    - best is to make your mind and use one conventions

- And read:

```
echo STR $STR
```

Exporting variables

- After an assignment **a="bla"** variables have the scope of the current script only

- If our intention is to make it available for programs/ scripts executed inside we need to export the variable:

```
A="bla"; export A
```

- or shorter:

```
export A="bla"
```

Predefined variables

- $HOME – the home directory
- $PWD – current directory
- $USER – login
- $PS1 – command line prompt
- $PATH – directories searched for commands
- All of them can be listed by command: env

Variable types

- String: **A="Alice has a cat"**
- Integer: **A=7** (Yes. We have changed the type of A!)
- Array: **A=("Alice", "in", "wonder")**
- When we want to declare/fix the type:
  - **declare "type" A**
    - eg: **declare –i A=9; A="blah"; echo $A**
    - -i (Integer) –f (Function) –r (Readonly) –t () –x(eXported) –a (Array)
    - -p (Print)
- constants: **readonly A=9**
  - useful for configuration parameters
  - issues error at an attempt of change A=10

Numerics

- The bash shell supports elementary operations on integers:
  - **A=$[4*B]** – most natural and suggested
  - **let A=B+C** – cleaner (imho)
  - **((A=B+4))** – most compact (for gurus)
- Possible operations +,-,*,/(only int), ++, --, +=, -=-,…
  - **((A++)), let A++**
  - **((A+=2)), let A+=2, A=$[A+2]**
- If we need to do one time floating point numerics: bc

```
echo 3.14*2+7.11 | bc
```

Strings

- Assignment:
  - `A="ala ma kota"`
  - `A=$B` –copy
  - Update == re-assignment (here with the concatenation):

```
export PATH=$PATH:$HOME/moje_skrypty/:.
```

- @@ a very good practice is to always use quotes "":
  - **Z=lux** – o.k.
  - **Z="lux torpedo"** – o.k,
    Z=lux torpedo -- error

Strings

DATA=data-from-2014-01-02

echo $DATA – listing

~~**echo $DATAtext**~~ ??? → **echo ${DATA}text** – join with some text {}

${#DATA} → number of characters

${DATA:7} → all from the character 7

echo ${DATA#data-from-} → removes sub-string (from the beginning if matching) % works at the end – wildcards are also possible **echo ${DATA#*-*-}**

i %% denote matching longest possible sub-string

${DATA/-/_} → replace sub-string (only the first occurence)

${DATA//-/_} → all occurrences

In all examples the DATA variable remains unchanged.

To change: DATA=${DATA…..}

Arrays

- Not quite convenient
  - To many arrays in the scripts usually calls for rewrite (presumably in python)
- Declaration:
  - **A = (Alice has a cat)**
  - **A=(”Alice” ”has” ”a cat”)** - better
  - **echo ${A[*]}** – the whole array printout, (yes. quite ugly)
  - **echo ${A[1]}** – a single element
  - single element assignment:
    - **A[1]=”had”**

Special variables

- $@ - script command line (i.e. how it was invoked)
- $1 – 1st argument, $2 – 2nd etc., $0 the name of the script
- **$#** - arguments count
- **$?** – execution status for the last program
  - eg: **diff a b > /dev/null; DIFFER=$?**
- **$$** - PID of the shell (a good unique ID) + …
- **$!** – PID of last called program (can be used to send it a kill signal)

Listing (of variables)

- **echo** – prints to stdout
  - **echo "dear $USER, type the password please"**
  - a useful option **–n** (no new line)

- **printf** – formatted output
  - **printf "%06d %25s" 56 "New York"**
  - Formatting flags exactly like in C stdio

Reading

- **read COUNT CAT**
  - reads from the keyboards and places characters in variables COUNT and CAT
  - options:
    - **–a** - reads to a table
    - **-p “>>>> ”** – with the prompt “>>>> ”
    - **-d** , - defines a delimiter (useful for tables)
    - **-s** – silent input (eg. for password)
    - **-t** – wait for the input for a defined time

Regular expressions

- Wildcards:
  - **.** – any character, \. - literally the dot
  - **[a-z]** – range of characters, ^[a-z] – outside of that range
  - **[[:digit:]]** - class of characters
  - other characters are literals – match exactly
  - special:
    - **^, $** - beginning and end of the line
    - **\<, \>** - beginning and end of the word
  - **()** -grupa

- Repeats:
  - * - 0 or more times
  - ? – 0 or 1
  - + 1 or more
  - {3} – exactly 3 times
  - {3,} – 3 or more
  - {3-7} – between 3 and 7 times

3, 7 –are just examples

[[:digit:]]{11} – PESEL

[A-Z]{3}[[:digit:]]{6} – ID

.*@.*\..* - e-mail

^$ - empty string

Larger script


```
#!/bin/bash
set -x # turn on debugging
source ~/.conf # load configuration,
# variables from the .conf available in this script
set +x # turn off debugging

mkdir include src run # create needed directories

PROJFILE=$1.pro # new variable (build from command)
cat $PROJ/projfile.pro $PROJFILE # copy some file

echo “PROJDIR=“$PWD >> $PROJFILE # append to the
   config file
 echo “To start run: source $PROJFILE”
```

Build instructions and external programs

- Execution of internal shell commands and external programs is indistinguishable
- However: external programs have to be in one of directories pointed by the PATH → `echo $PATH`, or we have to use a full path e.g: `./runme`

How they can invoked:

- cd – by naming it $? contains the status of the variable
- **cd; pwd; cd -; pwd** – several commands
- **xterm &** - in the background **$!** (PID)

```
xterm &
PIDTERM=$!
echo “Kill it? yes/no”
read DEC;
if test $DEC = “yes”; then
   kill $PIDTERM;
 fi
```

More on commands invocation

- `( cd; ls )` – commands in () executed in another shell, eg.: after this commands we do not change to the home directory
  - They can also be run in a background & → **(..)&**

- **{ cd; ls; }** – executes command in the current shell (like a function)

- Command substitution
  - When we need a value which can be obtained by executing some program eg. first line of the file conf.txt:

  head -1 conf.txt would print what we want

  `PROJ=\`head -1 conf\`` → replaces command with the result of thereof
    - alternative notion (modern) `PROJ=$(head -1 conf)`

Bash control instructions

test

- With conditional instructions the program `test`, is useful to convert expressions to boolean

- It is aliased also as [last argument has to be then] (to obtain logical syntax **[-f file])**
  - The test command has multiple options (see man test)
  - test blah → true (value 0)
  - test → false (value != 0)
  - files:
    - **-e XXX** → the file XXX exists
    - **-f XXX** → the file XXX is a plain file, -d →is directory
    - **X1 –nt X1** → X1 newer than X2
  - Strings:
    - **-z $str** → str is empty string
    - **-n $str** → non-empty string
    - **$A = $B** → two strings are the same (@@ Note = not == !)
    - **$A != $B** → two strings re different
  - Numbers:
    - **$a –gt 7** → identical **$a > 7**, …. **-ge, -eq, -lt, -le**

The if conditional

```
if test -d data; then
   echo “the directory exists”
elif test -f data; then
   echo “ it is a plain file”
else
   echo “does not exist”
fi #it is not a mistake, if -> fi
```

case

```
read UOPT

case $UOPT in

  quit|exit|stop|break) echo “end”; exit
  0;; #

  cont) echo “starting next iteration”;
  ITER=$[ITER+1]; runme $ITER ;;

 *) echo “do not know what to do”;;

esac # case -> esac :-)
```

The ”for” loop

for f in $TAB; then

echo $f

done # would be nice to have roof here :-), you can add it
 # > alias rof=done

- With the command substitution

```
for f in `ls *txt`; do mv $f ~/somewhere/; done
```

Iterating over the sequence of numbers with the seq:

```
for i in `seq 10`; do echo “Repeating: $i”; done
```

The while & until

```
while test; do

done
```

```
until test; do

done
```

additional useful commands: `break i continue`

sleep X → sleep X seconds

Functions

```
function list() {
    echo -n “---”
     echo $1
}
# somewhere later in the script
list “ala”
>> ---ala
```

@@ functions from sourced script **become like shell commands**
declare -f to list them

External programs

Printing text files

- cat → entire content to stdout, cat plik | progX – or more
- tac → quiz
- **less** → interactive browsing, one screen at a time
  - **zless** → browsing compressed text files
  - less contains quite extended browsing functionality
- **head** → 10 first lines
  - -X → x first lines lines
- **tail** → --||--

Filtering lines: grep

- **grep ERROR log** → list only lines containing ERROR
- **grep –e ERROR –e WARNING –e FATAL log** → with **e**ither of ERROR, WARNING or FATAL

- useful options:
  - **-A X –B X** – X lines **a**fter & **b**efore
  - **-H** – list the file name in addition
    grep bla *.txt
  - **-v** invert selection

Change characters in a stream

- **tr** → translate, reads stdin prints stdout
  - `tr "abc" "ABC"` → changes all characters from the first set to the second a → A, b → B itd.
  - most frequently used with special characters
    - e.g.: `tr ":" "\n"` → break to new line everywhere we have :

```
echo $PATH | tr ":" "\n"
```

useful options:

- **-s "X"**- squeze - replace repeated X by a single X
  - **cat file | tr –s "\n"** → only single empty lines
- **-d "X"**- delete -
  - **cat file | tr –d "\n"** →result has only one line

Extracting fragments of the line

- **cut** –
- **-d" "** – delimiter – the space in most of the cases …
- **-f1-6** – fields from 1 to 6 …
  - e.g.: all env varaibles

```
env | cut -d = -f1
```

- **-b0-12** - pick range of characters

awk

- “Very expanded” cut :-)
- matching lines and performs computation on them

`awk ‘/ERROR/ {print “>>> ” $0}’` → all lines containing ERROR have extra >>> prefix

- One can omit matching e.g.: perform action on each line
- **awk ‘{print $1 $3}’**
- a special sections BEGIN & END allow to perform actions the beginning and the end

```
awk ‘BEGIN {sum=0} {sum +=$4 } END {print “sum “ $sum}’
```

awk cd

- line fields delimiter can be redefined

BEGIN {FS=":"; OFS=","; ORS="\n\n" }

- **-f** - reads the awk commands from a file

- print is only one of commands, awk implements complete language ~C
  - contains controls: if, for, for .. in arrayX, while
  - many commands for strings processing:
    - length($1), substr($3, 1, 5), index($4, "kawalek"), match($x, "bla"), split, sub – substitute,
    - allows operations on fps
      - has few math functions

```
echo 2 3.5 | awk '{ print sqrt($2)**$1 }' -
```

Stream editor sed

- quite similar to awk but oriented towards processing in the stream
- Similar mechanics: chunks of the input read into the buffer: “pattern space” and then processed by the script
  - There exists also the “hold space” where the “pattern space” can be temporarily saved
- pattern space is searched for an „address” → followed by set of operations on the address

Syntax of sed scripts

- Addressing
  - adres to:
    - **X** – number, matches to line X
    - **$** - end of file
    - **/X/** - regex
    - **,** - begin end address delimiter
  - Command**:**
    - **p** – print
    - **q** – quit
    - **d** – delete

- Useful options:
  - **-e** – script in the command line
  - **-f** – script in the file
  - **-n** – do not print this patterns paces in which the pattern not found

- e.g.: extract what is between the HLT tags “body”:

```
sed -n -e '/body/,/\/body/ p’ index.html
```

sed: substitution

- command **s** means „replace”
  - most frequently used operation
  - e.g.: `echo Alice has a cat | sed -e 's/has/had/’`
  - Together with regexes quite powerful command
  - example:

```
sed  -e 's/head/HEAD/' -e 's/body/BODY/'  index.html
sed -i back -e 's/(/( /g' -e 's/)/ )/g'  pcut.C
cat Density.html | sed -n -e '/\<body/,/\<\/body/ p' |
  sed -n -e '/\<table/,/\<\/table/ p' | sed 's/\<table/
  ^M<table/' | tr "\n" " " | tr "^M" "\n"  | grep
  Density | tail -1
```

http://www.grymoire.com/Unix/Sed.html

Varia

- wget http://fis.agh.edu.pl/ - pull content from the webpage
  - curl –similar
- gnuplot –plotting from command line

Where to find examples?

- Many linux „programs” are in fact shell scripts

```
for d in `echo $PATH|tr : " "`; do file
  $d/*; done | grep shell
```

Where to look for help?

- The man + google/stack overflow are your friend