Linux e Bash
(rinforzino)

Matteo Duranti

matteo.duranti@infn.it

(cfr. https://www.uio.no/studier/emner/matnat/ifi/INF3331/h12/bash.pdf)

Linux

- **time** serve per mostrare le “timing statistics” di un altro comando/programma

[root@localhost shell]# time ./fibo_iterative.sh 15
The Fibonacci sequence for the number 15 is :
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610

real 0m0.008s
user 0m0.008s
sys 0m0.000s

Overview of Unix shells

- The original scripting languages were (extensions of) command interpreters in operating systems
- Primary example: Unix shells
- Bourne shell (`sh`) was the first major shell
- C and TC shell (`csh` and `tcsh`) had improved command interpreters, but were less popular than Bourne shell for programming
- Bourne Again shell (Bash/`bash`): GNU/FSF improvement of Bourne shell
- Other Bash-like shells: Korn shell (`ksh`), Z shell (`zsh`)
- Bash is the dominating Unix shell today

Why learn Bash? (2)

- Shell scripts evolve naturally from a workflow:
  1. A sequence of commands you use often are placed in a file
  2. Command-line options are introduced to enable different options to be passed to the commands
  3. Introducing variables, if tests, loops enables more complex program flow
  4. At some point pre- and postprocessing becomes too advanced for bash, at which point (parts of) the script should be ported to Python or other tools
- Shell scripts are often used to glue more advanced scripts in Perl and Python

Scientific Hello World script

- Let's start with a script writing "Hello, World!"
- Scientific computing extension: compute the sine of a number as well
- The script (hw.sh) should be run like this:

  ```
  ./hw.sh 3.4
  ```

  or (less common):

  ```
  bash hw.sh 3.4
  ```

- Output:

  ```
  Hello, World! sin(3.4)=-0.255541102027
  ```

- Can be done with a single line of code:

  ```
  echo "Hello, World! sin($1)=$(echo "s($1)" | bc -l)"
  ```

Purpose of this script

Demonstrate

- how to read a command-line argument
- how to call a math (sine) function
- how to work with variables
- how to print text and numbers

Remark

- We use plain Bourne shell (`/bin/sh`) when special features of Bash (`/bin/bash`) are not needed
- Most of our examples can in fact be run under Bourne shell (and of course also Bash)
- Note that Bourne shell (`/bin/sh`) is usually just a link to Bash (`/bin/bash`) on Linux systems
  (Bourne shell is proprietary code, whereas Bash is open source)

The code, in extended version

File hw.sh:

shbang

```
#!/bin/sh
r=$1  # store first command-line argument in r
s=‘echo "s($r)" | bc -l‘

# print to the screen:
echo "Hello, World! sin($r)=$s"
```

Comments

- The first line specifies the interpreter of the script (here `/bin/sh`, could also have used `/bin/bash`)
- The command-line variables are available as the script variables

  `$1 $2 $3 $4 and so on`

- Variables are initialized as

  `r=$1`

  while the *value* of `r` requires a dollar prefix:

  ```
  my_new_variable=$r  # copy r to my_new_variable
  ```

Bash and math

- Bourne shell and Bash have very little built-in math, we therefore need to use bc, Perl or Awk to do the math

```
s=‘echo "s($r)" | bc -l‘
s=‘perl -e ’$s=sin($ARGV[0]); print $s;’ $r‘
s=‘awk "BEGIN { s=sin($r); print s;}"‘
# or shorter:
s=‘awk "BEGIN {print sin($r)}"‘
```

- Back quotes means executing the command inside the quotes and assigning the output to the variable on the left-hand-side

```
some_variable=‘some Unix command‘

# alternative notation:
some_variable=$(some Unix command)
```

The bc program

- bc = interactive calculator
- Documentation: man bc
- bc -l means bc with math library
- Note: sin is s, cos is c, exp is e
- echo sends a text to be interpreted by bc and bc responds with output (which we assign to `s`)

```
variable=‘echo "math expression" | bc -l‘
```

Printing

- The `echo` command is used for writing:

  ```
  echo "Hello, World! sin($r)=$s"
  ```

  and variables can be inserted in the text string (variable interpolation)
- Bash also has a printf function for format control:

  ```
  printf "Hello, World! sin(%g)=%12.5e\n" $r $s
  ```

- `cat` is usually used for printing multi-line text (see next slide)

Convenient debugging tool: -x

- Each source code line is printed prior to its execution of you -x as option to `/bin/sh` or `/bin/bash`
- Either in the header

  ```
  #!/bin/sh -x
  ```

  or on the command line:

  ```
  unix> /bin/sh -x hw.sh
  unix> sh -x hw.sh
  unix> bash -x hw.sh
  ```

- Very convenient during debugging

Parsing command-line options

```
# read variables from the command line, one by one:
while [ $# -gt 0 ]  # $# = no of command-line args.
do
    option = $1; # load command-line arg into option
    shift;       # eat currently first command-line arg
    case "$option" in
        -m)
            m=$1; shift; ;;  # load next command-line arg
        -b)
            b=$1; shift; ;;
        ...
        *)
            echo "$0: invalid option \"$option\""; exit ;;
    esac
done
```

Alternative to case: if

`case` is standard when parsing command-line arguments in Bash, but if-tests can also be used. Consider

```
case "$option" in
    -m)
        m=$1; shift; ;;  # load next command-line arg
    -b)
        b=$1; shift; ;;
    *)
        echo "$0: invalid option \"$option\""; exit ;;
esac
```

versus

```
if [ "$option" == "-m" ]; then
    m=$1; shift;  # load next command-line arg
elif [ "$option" == "-b" ]; then
    b=$1; shift;
else
    echo "$0: invalid option \"$option\""; exit
fi
```

Creating a subdirectory

```
dir=$case
# check if $dir is a directory:
if [ -d $dir ]
  # yes, it is; remove this directory tree
  then
    rm -r $dir
fi
mkdir $dir   # create new directory $dir
cd $dir      # move to $dir

# the 'then' statement can also appear on the 1st line:
if [ -d $dir ]; then
  rm -r $dir
fi

# another form of if-tests:
if test -d $dir; then
  rm -r $dir
fi

# and a shortcut:
[ -d $dir ] && rm -r $dir
test -d $dir && rm -r $dir
```

Writing an input file

'Here document' for multi-line output:

```
# write to $case.i the lines that appear between
# the EOF symbols:

cat > $case.i <<EOF
        $m
        $b
        $c
        $func
        $A
        $w
        $y0
        $tstop
        $dt
EOF
```

Remark (1)

- Variables can in Bash be integers, strings or arrays
- For safety, declare the type of a variable if it is not a string:

```
declare -i i   # i is an integer
declare -a A   # A is an array
```

Remark (2)

- Comparison of two integers use a syntax different comparison of two strings:

```
if [ $i -lt 10 ]; then         # integer comparison
if [ "$name" == "10" ]; then   # string  comparison
```

- Unless you have declared a variable to be an integer, assume that all variables are strings and use double quotes (strings) when comparing variables in an if test

```
if [ "$?" != "0" ]; then  # this is safe
if [  $?  !=  0  ]; then  # might be unsafe
```

Some common tasks in Bash

- file writing
- for-loops
- running an application
- pipes
- writing functions
- file globbing, testing file types
- copying and renaming files, creating and moving to directories, creating directory paths, removing files and directories
- directory tree traversal
- packing directory trees

File writing

```
outfilename="myprog2.cpp"

# append multi-line text (here document):
cat >> $filename <<EOF
/*
  This file, "$outfilename", is a version
  of "$infilename" where each line is numbered.
*/
EOF

# other applications of cat:
cat myfile              # write myfile to the screen
cat myfile >  yourfile # write myfile to yourfile
cat myfile >> yourfile # append myfile to yourfile
cat myfile | wc         # send myfile as input to wc
```

For-loops

- The for element in list construction:

```
files='/bin/ls *.tmp'
# we use /bin/ls in case ls is aliased

for file in $files
do
  echo removing $file
  rm -f $file
done
```

- Traverse command-line arguments:

```
for arg; do
  # do something with $arg
done

# or full syntax; command-line args are stored in $@
for arg in $@; do
  # do something with $arg
done
```

Counters

- Declare an integer counter:

```
declare -i counter
counter=0
# arithmetic expressions must appear inside (( ))
((counter++))
echo $counter  # yields 1
```

- For-loop with counter:

```
declare -i n; n=1
for arg in $@; do
  echo "command-line argument no. $n is <$arg>"
  ((n++))
done
```

C-style for-loops

```
declare -i i
for ((i=0; i<$n; i++)); do
  echo $c
done
```

Numerical expressions

Numerical expressions can be evaluated using bc:

```
echo "s(1.2)" | bc -l  # the sine of 1.2
# -l loads the math library for bc

echo "e(1.2) + c(0)" | bc -l  # exp(1.2)+cos(0)

# assignment:
s=‘echo "s($r)" | bc -l‘

# or using Perl:
s=‘perl -e "print sin($r)"‘
```

Functions

```
# compute x^5*exp(-x) if x>0, else 0 :

function calc() {
   echo "
   if ( $1 >= 0.0 ) {
      ($1)^5*e(-($1))
   } else {
      0.0
   } " | bc -l
}

# function arguments: $1 $2 $3 and so on
# return value: last statement

# call:
r=4.2
s=‘calc $r‘
```

Another function example

```
#!/bin/bash

function statistics {
  avg=0; n=0
  for i in $@; do
    avg=‘echo $avg + $i | bc -l‘
    n=‘echo $n + 1 | bc -l‘
  done
  avg=‘echo $avg/$n | bc -l‘

  max=$1; min=$1; shift;
  for i in $@; do
    if [ ‘echo "$i < $min" | bc -l‘ != 0 ]; then
      min=$i; fi
    if [ ‘echo "$i > $max" | bc -l‘ != 0 ]; then
      max=$i; fi
  done
  printf "%.3f %g %g\n" $avg $min $max
}
```

Calling the function

```
statistics 1.2 6 -998.1 1 0.1

# statistics returns a list of numbers
res=‘statistics 1.2 6 -998.1 1 0.1‘

for r in $res; do echo "result=$r"; done

echo "average, min and max = $res"
```

funzioni in bash - ricorsività

- **ricorsione** come più o meno in tutti i linguaggi di programmazione è possibile fare funzioni che richiamano sé stesse:

```
#!/bin/bash

fact ()
{
  local number=$1
  #  Variable "number" must be declared as local,
  #+ otherwise this doesn't work.
  if [ "$number" -eq 0 ]
  then
    factorial=1    # Factorial of 0 = 1.
  else
    let "decrnum = number - 1"
    fact $decrnum  # Recursive function call (the function calls itself).
    let "factorial = $number * $?"
  fi

  return $factorial
}

fact $1
echo "Factorial of $1 is $?."

exit 0
```

File globbing

- List all .ps and .gif files using wildcard notation:

```
files=‘ls *.ps *.gif‘

# or safer, if you have aliased ls:
files=‘/bin/ls *.ps *.gif‘

# compress and move the files:
gzip $files
for file in $files; do
  mv ${file}.gz $HOME/images
```

Testing file types

```
if [ -f $myfile ]; then
    echo "$myfile is a plain file"
fi

# or equivalently:
if test -f $myfile; then
    echo "$myfile is a plain file"
fi

if [ ! -d $myfile ]; then
    echo "$myfile is NOT a directory"
fi

if [ -x $myfile ]; then
    echo "$myfile is executable"
fi

[ -z $myfile ] && echo "empty file $myfile"
```

Successione di Fibonacci

Successione di Fibonacci

Da Wikipedia, l'enciclopedia libera.

Questa voce o sezione sull'argomento matematica è ritenuta da controllare.
Motivo: *È stato inserito parecchio materiale mal formattato, con notazioni diverse rispetto al resto della voce, alcuni passi sono poco contestualizzati e di altri non si coglie la rilevanza*

Partecipa alla discussione e/o correggi la voce. Segui i suggerimenti del progetto di riferimento.

In matematica, la **successione di Fibonacci**, indicata con F_n o con $Fib(n)$, è una successione di numeri interi positivi in cui ciascun numero è la somma dei due precedenti e i primi due termini della successione sono per definizione $F_1 = 1$ e $F_2 = 1$. Tale successione ha quindi una definizione ricorsiva secondo la seguente regola:

$$F_1 = 1,$$
$$F_2 = 1,$$
$$F_n = F_{n-1} + F_{n-2} \text{ (per ogni n>2)}$$

Gli elementi F_n sono anche detti *numeri di Fibonacci*.

I primi termini della successione di Fibonacci sono: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

La successione prende il nome dal matematico pisano del XIII secolo Leonardo Fibonacci.

Esercitazione

- creare uno script in bash che calcoli la successione di Fibonacci senza utilizzare la ricorsione;
- creare uno script in bash che calcoli la successione di Fibonacci utilizzando la ricorsione;
- confrontare il tempo di esecuzione dei due script precedenti (i.e. *time*);
- creare uno script in bash che crei una directory con all'interno *N* (passato come parametro) sottodirectory chiamate *dir_1, dir_2, dir_3, …, dir_N* e all'interno di ognuna ci siano *M* (dove *M* è dato da *dir_M*) file di testo, chiamati *el_1, el_2, el_3, …, el_M,* ciascuno contenente (scritto nel file di testo) il valore del *n*-esimo coefficiente della successione di Fibonacci (*n* dato da *el_n*);
- rinominare tutti i file tenendo conto del nome file col massimo numero, in modo che abbiano tutti lo stesso numero di cifre: se in *dir_12* l'"ultimo" file è *el_144* (la serie di Fibonacci fino a F_{12} è 1,1,2,3,5,8,13,21,34,55,89,144), allora anche *el_1* dovrà essere rinominato *el_001, el_13* → *el_013*, etc…