Slides from INF3331 lectures - Bash programming

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

Basic Bash programming

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?

- Learning Bash means learning Unix
- Learning Bash means learning the roots of scripting (Bourne shell is a subset of Bash)
- Shell scripts, especially in Bourne shell and Bash, are frequently encountered on Unix systems
- Bash is widely available (open source) and the dominating command interpreter and scripting language on today's Unix systems

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

More information

- `man bash`
- “Introduction to and overview of Unix” link in `doc.html`

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:

```
#!/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

File reading and writing

- Bourne shell and Bash are not much used for file reading and manipulation; usually one calls up Sed, Awk, Perl or Python to do file manipulation
- File writing is efficiently done by 'here documents':

```
cat > myfile <<EOF
multi-line text
can now be inserted here,
and variable interpolation
a la $myvariable is
supported. The final EOF must
start in column 1 of the
script file.
EOF
```

Simulation and visualization script

- Typical application in numerical simulation:
  - run a simulation program
  - run a visualization program and produce graphs
- Programs are supposed to run in batch
- Putting the two commands in a file, with some glue, makes a classical Unix script

Setting default parameters

```
#!/bin/sh

pi=3.14159
m=1.0; b=0.7; c=5.0; func="y"; A=5.0;
w=`echo 2*$pi | bc`
y0=0.2; tstop=30.0; dt=0.05; case="tmp1"
screenplot=1
```

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
```

Running the simulation

- Stand-alone programs can be run by just typing the name of the program
- If the program reads data from standard input, we can put the input in a file and *redirect input*:

```
oscillator < $case.i
```

- Can check for successful execution:

```
# the shell variable $? is 0 if last command
# was successful, otherwise $? != 0

if [ "$?" != "0" ]; then
  echo "running oscillator failed"; exit 1
fi

# exit n sets $? to n
```

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
```

Making plots

- Make Gnuplot script:

```
echo "set title '$case: m=$m ...'" > $case.gnuplot
...
# contiune writing with a here document:
cat >> $case.gnuplot <<EOF
set size ratio 0.3 1.5, 1.0;
...
plot 'sim.dat' title 'y(t)' with lines;
...
EOF
```

- Run Gnuplot:

```
gnuplot -geometry 800x200 -persist $case.gnuplot
if [ "$?" != "0" ]; then
  echo "running gnuplot failed"; exit 1
fi
```

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
```

Example: bundle files

- Pack a series of files into one file
- Executing this single file as a Bash script packs out all the individual files again (!)
- Usage:

```
bundle file1 file2 file3 > onefile  # pack
bash onefile # unpack
```

- Writing `bundle` is easy:

```
#/bin/sh
for i in $@; do
    echo "echo unpacking file $i"
    echo "cat > $i <<EOF"
    cat $i
    echo "EOF"
done
```

The bundle output file

- Consider 2 fake files; file1

  ```
  Hello, World!
  No sine computations today
  ```

  and file2

  ```
  1.0 2.0 4.0
  0.1 0.2 0.4
  ```

- Running `bundle file1 file2` yields the output

  ```
  echo unpacking file file1
  cat > file1 <<EOF
  Hello, World!
  No sine computations today
  EOF
  echo unpacking file file2
  cat > file2 <<EOF
  1.0 2.0 4.0
  0.1 0.2 0.4
  EOF
  ```

Running an application

- Running in the foreground:

```
cmd="myprog -c file.1 -p -f -q";
$cmd < my_input_file

# output is directed to the file res
$cmd < my_input_file > res

# process res file by Sed, Awk, Perl or Python
```

- Running in the background:

```
myprog -c file.1 -p -f -q < my_input_file &
```

  or stop a foreground job with Ctrl-Z and then type `bg`

Pipes

- Output from one command can be sent as input to another command via a pipe

```
# send files with size to sort -rn
# (reverse numerical sort) to get a list
# of files sorted after their sizes:

/bin/ls -s | sort -r


cat $case.i | oscillator
# is the same as
oscillator < $case.i
```

- Make a new application: sort all files in a directory tree `root`, with the largest files appearing first, and equip the output with paging functionality:

```
du -a root | sort -rn | less
```

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"
```

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"
```

Rename, copy and remove files

```
# rename $myfile to tmp.1:
mv $myfile tmp.1

# force renaming:
mv -f $myfile tmp.1

# move a directory tree my tree to $root:
mv mytree $root

# copy myfile to $tmpfile:
cp myfile $tmpfile

# copy a directory tree mytree recursively to $root:
cp -r mytree $root

# remove myfile and all files with suffix .ps:
rm myfile *.ps

# remove a non-empty directory tmp/mydir:
rm -r tmp/mydir
```

Directory management

```
# make directory:
$dir = "mynewdir";
mkdir $mynewdir
mkdir -m 0755 $dir  # readable for all
mkdir -m 0700 $dir  # readable for owner only
mkdir -m 0777 $dir  # all rights for all

# move to $dir
cd $dir
# move to $HOME
cd

# create intermediate directories (the whole path):
mkdirhier $HOME/bash/prosjects/test1
# or with GNU mkdir:
mkdir -p  $HOME/bash/prosjects/test1
```

The find command

Very useful command!

- `find` visits all files in a directory tree and can execute one or more commands for every file
- Basic example: find the `oscillator` codes

```
find $scripting/src -name 'oscillator*' -print
```

- Or find all PostScript files

```
find $HOME \( -name '*.ps' -o -name '*.eps' \) -print
```

- We can also run a command for each file:

```
find rootdir -name filenamespec -exec command {} \; -print
# {} is the current filename
```

Applications of find (1)

- Find all files larger than 2000 blocks a 512 bytes (=1Mb):

```
find $HOME -name '*' -type f -size +2000 -exec ls -s {} \;
```

- Remove all these files:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \; -exec rm -f {} \;
```

  or ask the user for permission to remove:

```
find $HOME -name '*' -type f -size +2000 \
     -exec ls -s {} \; -ok rm -f {} \;
```

Applications of find (2)

- Find all files not being accessed for the last 90 days:

  ```
  find $HOME -name '*' -atime +90 -print
  ```

  and move these to /tmp/trash:

  ```
  find $HOME -name '*' -atime +90 -print \
       -exec mv -f {} /tmp/trash \;
  ```

- Note: this one does seemingly nothing...

  ```
  find ~hpl/projects -name '*.tex'
  ```

  because it lacks the `-print` option for printing the name of all *.tex files (common mistake)

Tar and gzip

- The `tar` command can pack single files or all files in a directory tree into one file, which can be unpacked later

```
tar -cvf myfiles.tar mytree file1 file2

# options:
# c: pack, v: list name of files, f: pack into file

# unpack the mytree tree and the files file1 and file2:
tar -xvf myfiles.tar

# options:
# x: extract (unpack)
```

- The tarfile can be compressed:

```
gzip mytar.tar

# result: mytar.tar.gz
```

Two find/tar/gzip examples

- Pack all PostScript figures:

```
tar -cvf ps.tar ‘find $HOME -name ’*.ps’ -print‘
gzip ps.tar
```

- Pack a directory but remove CVS directories and redundant files

```
# take a copy of the original directory:
cp -r myhacks /tmp/oblig1-hpl
# remove CVS directories
find /tmp/oblig1-hpl -name CVS -print -exec rm -rf {} \;
# remove redundant files:
find /tmp/oblig1-hpl \( -name ’*~’ -o -name ’*.bak’ \
 -o -name ’*.log’ \) -print -exec rm -f {} \;
# pack files:
tar -cf oblig1-hpl.tar /tmp/tar/oblig1-hpl.tar
gzip oblig1-hpl.tar
# send oblig1-hpl.tar.gz as mail attachment
```