The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific and
Cultural Organization

IAEA
International Atomic Energy Agency

2057-8

First Workshop on Open Source and Internet Technology for Scientific Environment: with case studies from Environmental Monitoring

7 - 25 September 2009

Shell Programming

Paul Bartholdi

Observatoire de Geneve
Chemin des Maillettes 51
CH-1290 Sauverny
Switzerland

1 Shell programming

When a set of commands is repeated more than 2 or 3 times, then it is usually worth putting them into a file and executing the file, passing possibly parameters. Such files are called script files in UNIX.

All UNIX shells offer lots of usual *programming constructs*, as variables, conditionals and loops, input and output, and even some rudimentary arithmetic. Shell programming cannot replace C programming, in particular it is much slower, but it can be very effective in organizing together the repetitive and possibly conditional execution of programs.

Writing script files can have two other advantages:

– they can be edited until they work, even once ...

– they keep track of what was done, either as a log, or as an example for a similar problem in the future.

To be executable, a file just needs the `x` bit set in its permissions. This is done with the `chmod +x` *script* command.

As many different shells can be used in UNIX, it is preferable to add as a first line a comment indicating to the system which shell is used. So the first line of a script file should look like `#!/bin/sh` or whatever other shell is used (remember they have different syntax, and should not be confused).

1.1 Comments

Any character between the `#` and the end-of-line is treated as a comment. The example just above is really a comment, and is understood by the shell as a possible indication about which shell should be used. In such a case, the `#` is called the *magic number*.

1.2 Quotes

Two types of quote symbols can be used: ' and ".

Inside ' ', no special characters are interpreted.

Inside " ", only $, `, !, and \ are interpreted.

Any special character can be transformed into a normal one with a \ in front.

Try:

```
Test="NoGood"
echo 1. Test           # just ascii string
echo 2. $Test          # $    in front
echo 3. \$Test         # \$   in front
echo 4. \\$Test        # \\$  in front
```

1.3 Parameter passing

A command can be followed by parameters as "words" separated by spaces or tabs. The end-of-line, a ;, redirections or pipes, end the command.

Inside a script, $n, where n is a digit, will be replaced by the corresponding parameter. Notice that $0 corresponds to the name of the command itself.

As a very simple example, here is a script that will compile a C program, and execute it immediately. The name of the program is passed to the script as a parameter.

```
#!/bin/sh -x
gcc -O3 -o $1 $1.c
$1
```

To compile and execute *threads.c*, one would type `ccc threads` where `ccc`

is the name of the script.

1.4 Variables

Variables can be defined inside a shell. Except if exported, they are not seen outside the shell. Variable names are made of letters, digits and underscores only, starting with a letter or an underscore.

They can be defined with =, without any spaces around the = sign, or read from the terminal or a file, as in the following:

```
Test="Order==$1"
read answer
```

and used, as for parameters, with a $ in front for them to be replaced with their content.

```
if [ "x$answer" = "xY" ]; then
   SetPower $level
fi
select "$Test"
```

1.5 Environment variables PATH, MANPATH and LD_LIBRARY_PATH

When the name of a program (a file name effectively) is given for execution, the system will look in successive directories, and execute the first one found.

In the same way, man looks in successive directories and prints the first corresponding pages found, and the loader looks in the list of directories for dynamic libraries.

These lists of directories are given in PATH , MANPATH and LD_LIBRARY_PATH .

The directory names are separated by colon (“ : ”) characters.

To add a new directory, use command (in *bash*):

```
PATH=${PATH}: <my_dir>
```

or

```
PATH= <my_dir>:${PATH}
```

The first version puts the new directory at the end, and the second in front, of the list. Both versions have some advantages.

`tcsh` keeps a hash table of all executables found in the `PATH` . This table is setup at login, but it is not automatically updated when `PATH` changes. The command `rehash` can be used to update manually the hash table.

- a “generous” `PATH` is predefined in most Linux systems
- the current directory “ `.` ” is usually part of the `PATH` . It is better to put it at the end of the list to avoid replacing a system program.
- you can put all your executables in a directory called `~/bin` and add `~/bin` to your `PATH` (in the file `~/.login` or `~/.profile`).
- you can do the same for your personal `man` pages.
- to see the full `PATH` as defined now, use the command:

  ```
  echo $PATH
  ```

- to see all environment variables:

  ```
  env
  ```

- to find where an executable is:

  `which` *my_program*

- to find where are all copies of a program (in the list defined by `PATH`):

  `whereis` *your_program*

  You may have to redefine `whereis` in an alias to search the full `PATH` :

  ```
  alias=whereis "whereis -B $PATH -f"
  ```

- If you add directories in an uncontrolled way, the same directory may appear in different places ... To avoid this, you can use the program `envv` available in the public domain:

  `eval ‘envv add PATH` *my_dir* `1‘`

  The last number, if present, indicates the position of the new directory in the list. Without a number, the new directory is put at the right end of the list.

  Notice that `envv` is insensitive to the shell used (same syntax in `tcsh`, `bash` and `ksh`).

1.6 Reading data

Variables can be read from the keyboard with the `read` command as seen in section 1.4. Any file can be redirected to the standard input with the command `exec 0<file`. Then the `read` command gets lines form the file into the variables. The arguments can be individualy recovered with the `set` command:

```
exec 0< Classes
read head
set $head
echo The heads are: $1 $2 $3
```

1.7 Loop – `for` command

In `bash` , the command `for` permits to loop over many commands with a variable taking successive values from a list (see section 2.1 for a `csh` equivalent).

The syntax is:

`for` <*variable name*> `in` <*list of values*> `; do`
<*commands*>

<*commands*>
...
done

Here are a few examples using `for` in `bash` scripts. You may want to try to rewrite them for `csh`.

1. Repeat 10 times a benchmark:

   ```
   for bench in 1 2 3 4 5 6 7 8 9 10 ; do
      echo Benchmark Nb: $bench
      benchmark | tee bench.log_$bench
   done
   ```

2. Doing `ftp` to a set of machines. We assume that the commands for `ftp` have been prepared in a file `ftp.cmds` :

   ```
   for station in 1 2 3 7 13 19 27 ; do
      echo "Connecting to station infolab-$station"
      ftp infolab-$station < ftp.cmds
   done
   ```

   Such commands enable us to update a lot of stations in a relatively easy way.

1.8 File name modifiers

The variable names can be modified with the following modifiers:

<*variable name*>`:r` suppresses all the possible suffixes.

<*variable name*>`:s/` <*old*>`/` <*new*>`/` substitutes <*new*> for <*old*>.

Many more modifiers exist; man pages of `csh` gives a complete list.

The following code saves all executables and recompiles:

```
for file in *.c ; do
   echo $file
   cp $file:r $file:r_org
   gcc -g -o $file:r $file
done
```

2 Shell Programming

2.1 bash and csh command syntax compared

Today, many people use tcsh for interactive work. Others prefer bash or ksh . They have so many goodies. But for shell programming, writing scripts, the choice is really open between sh and its offsprings (ksh, bash ...) on one side, and csh on the other. ksh or bash are now the default standard on Linux. They are simpler yet most powerful of all. csh on the other hand has the advantage of being a subset of tcsh, with which the user is probably more comfortable. As with many other choices with computers, it has become a question of religion. The choice is really very much yours!

If your problem is very complex, requiring you to handle arrays or to manipulate many files, then probably neither bash nor csh is sufficient.

awk is an ideal tool to manipulate text in any form, but it is not really intended for shell programming. It has only few interactions with the system, with the file system, etc.

perl provides almost everything you may ever wish, including, in the script language, all facilities of awk and sed, both indexed and context addressed arrays, etc. perl 5 is now available with most Linux distributions. As for tcsh, it is not part of the base system and has to be installed specifically by the "system manager".

Python is a rather different scripting language. It is fully object oriented, with a clean and simple syntax. It can be expanded extensively using libraries, written either in Python, in C or many other programming languages. As

with shells, it can be used both interactively and through script files. Moreover, code written in Python are probably the most readable.

Table 1 compares the main commands used in `bash` and `csh`. As you will see, some are missing in one or the other, others are definitely simpler in one, and many others are quite similar in both.

2.2 Signals used with shells

The main signals used in shells are: `INT` (2), `QUIT` (3), `KILL` (9), `TERM` (15), `STOP` (23) and `CONT` (25). `KILL` cannot be caught or ignored, and will bring your shell to an end. `STOP` and `CONT` allow to stop a shell (or any task) temporarily and then restart it without losing anything.

Here is a full list of signals as used in Linux. It is extracted from the file `/usr/src/linux/include/asm/signal.h`

```
#include <linux/types.h>

#define SIGHUP  1
#define SIGINT  2
#define SIGQUIT  3
#define SIGILL  4
#define SIGTRAP  5
#define SIGABRT  6
#define SIGIOT  6
#define SIGBUS  7
#define SIGFPE  8
#define SIGKILL  9
#define SIGUSR1 10
#define SIGSEGV 11
#define SIGUSR2 12
#define SIGPIPE 13
#define SIGALRM 14
#define SIGTERM 15
#define SIGSTKFLT 16
#define SIGCHLD 17
#define SIGCONT 18
#define SIGSTOP 19
```

Table 1: Comparison between `bash` and `csh`

bash	csh
Arithmetic	
`$(( ... ))`	`@`*var*`=`*expr*
`expr` *expression*	
Loops	

`for` *id* `in` *words* `; do`
list `;`
`done`
	`foreach` *var* `(` *words* `)`
	...
	`end`

Repeated command	
–	`repeat` *count command*
Menu input	

`select` *id* `in` *words* `;`
`do` *list* `;`
`done`
	–

Case	

`case` *word* `in`
pattern `)` *list* `;;`
pattern `)` *list* `;;`
`*` `)` *list* `;;`
`esac`
	`switch (` *string* `)`
	`case` *label* `:`
	...
	`breaksw`
	`default:`
	`endsw`

Conditionals	

`if` *list* `; then`
list `;`
`elif`
list `;`
`else`
list `;`
`fi`
	`if (`*expression* `) then`
	...
	`else if (`*expression* `) then`
	...
	`else`
	...
	`endif`

Conditional loops	

`while` *list* `; do`
list `;`
`done`
	`while (` *expression* `)`
	...
	`end`

`until` *list* `; do`
list `;`
`done`	

Function	
`function` *id* `() {` *list* `; }`	
Signal capture	
`trap` *command signal*	`onintr` *label*
Breaking loops	
`break`	`break` `continue`

```
#define SIGTSTP 20
#define SIGTTIN 21
#define SIGTTOU 22
#define SIGURG 23
#define SIGXCPU 24
#define SIGXFSZ 25
#define SIGVTALRM 26
#define SIGPROF 27
#define SIGWINCH 28
#define SIGIO 29
#define SIGPOLL SIGIO
/*
#define SIGLOST 29
*/
#define SIGPWR 30
#define SIGSYS 31
#define SIGUNUSED 31

/* These should not be considered constants from userland.  */
#define SIGRTMIN 32
#define SIGRTMAX (_NSIG-1)
```

2.3 Sample shell scripts

The following pages list some shell scripts that present various aspects of shell programming. Almost every construction is present, though not necessarily with every option. Some are just toy scripts (`calc`) whereas others are real programs used daily for system maintenance (`crlicense`, `png1` and `png2`). `flist` has been used to create this listing.

Table 3 shows commands and corresponding scripts in which they are used. The scripts below are in alphabetical order. Their names appear in the listing at the right, after a long dashed line separating the various scripts. They are written in `ksh` or `bash`, but are easily converted to `csh`.

Table 3: Commands and corresponding scripts

arithmetic	calc calc2 guess1 guess2 minutes
awk	KillKillMeAfter
loops	convert convert2 flist tolower toupper
select	term1 term2
case	convert minutes term2
if	KillKillMeAfter KillMeAfter convert ddmf_check filinfo flist grep2 guess1 guess2 term1 term2
while	calc2 convert guess1 guess2 minutes
function	convert3
trap	calc2 guess1

Sample listing

```
Tue Oct 3 11:41:33 MEST 2000
------------------------------------------------------------    KillKillMeAfter
#!/bin/ksh -f
# Kill the KillMeAfter started by pid $1
# Also kill the sleep started by KillMeAfter

GAWK=/usr/bin/gawk

KMApid=‘ps -ef             | \
  tr -s ’ ’                | \
  egrep KillMeAfter        | \
  egrep -v KillKillMeAfter | \
  egrep -v egrep           | \
  $GAWK -v pid=$1 ’$10 == pid { print $2  } ’ ‘

ps -ef        | \
  tr -s ’ ’   | \
  egrep sleep | \
  egrep -v egrep  > /tmp/KMA_$$

if [ -s /tmp/KMA_$$ ] && [ "X${KMApid}" != "X" ]  ; then
  sleeppid=‘cat /tmp/KMA_$$ | $GAWK -v pid=$KMApid ’$3 == pid { print $2  } ’  ‘
  \rm -f /tmp/KMA_$$
else
  \rm -f /tmp/KMA_$$
  exit 0
fi
```

```
### echo $$ : $KMApid / $sleeppid

if [ "X$KMApid" != "X" ] || [ "X$sleeppid" != "X" ]  ; then
  kill -9 $KMApid $sleeppid  2> /dev/null
fi

exit 0
-------------------------------------------------------    KillMeAfter
#!/bin/ksh
# called by some script, with  pid as parameter $1,
# expected to kill it after $2 sec

# echo $0 : pid=$1
# echo $0 go to sleep for $2 sec
sleep $2
# echo $0 weak up
if ‘ps -ef -o pid | egrep $1 > /dev/null ‘ ; then
  kill -9 $1
#   echo pid : $1 should be dead now
# else
#   echo pid : $1 was already killed
fi
exit 0
-------------------------------------------------------    calc
#!/bin/bash
# Very simple calculator - one expression per command

echo $(($*))
exit 0
-------------------------------------------------------    calc2
#!/bin/bash
# simple calculator, multiple expressions until ^C

trap ’echo Thank you for your visit ’ EXIT

while read expr’?expression ’; do
     echo $(($expr))
done
exit 0
-------------------------------------------------------    convert
#!/bin/bash
# convert tiff files to ps

echo there are $# files to convert :
```

```
echo $*
echo Is this correct ?

done=false
while [[ $done == false ]]; do
  done=true
  {
    echo 'Enter y for yes'
    echo 'Enter n for no'
  } >&2
  read REPLY?'Answer ?'
  case $REPLY in
    y ) GO=y ;;
    n ) GO=n ;;
    * ) echo '***** Invalid'
        done=falase ;;
  esac
done
if [[ "$GO" = y\"y" ]]; then
  for filename in "$@"; do
    newfile=${filename%.tiff}.ps
    eval convert $filename $newfile
  done
fi
exit 0
------------------------------------------------------------    convert2
#!/bin/bash
# simple program to convert tiff files into ps

for filename in "$@" ; do
  psfile=${filename%.tiff}.ps
  eval convert $filename $psfile
done
exit 0
------------------------------------------------------------    convert3
#!/bin/bash
# simple program to convert tiff files into ps

function tops {
  psfile=${1%.tiff}.ps
  echo $1 $psfile
  convert $1 $psfile
  }

for filename in "$@" ; do
```

```
  tops $filename
done
exit 0
------------------------------------------------------------    copro
#!/bin/bash
# coprocess in ksh

ed - memo |&
echo -p /world/
read -p search
echo "$search"
exit 0
------------------------------------------------------------    copro2
#!/bin/bash
# coprocess 2 in ksh

search=eval echo /world/ | ed - memo
echo "$search"
exit 0
------------------------------------------------------------    filinfo
#!/bin/bash
# print informations about a file

if [[ ! -a $1 ]] ; then
  echo "file $1 does not exist !"
  return 1
fi

if [[ -d $1 ]] ; then
  echo -n "$1 is a directory that you may"
  if [[ ! -x $1 ]] ; then
    echo -n " not "
  fi
  echo "search."
elif [[ -f $1 ]] ; then
  echo "$1 is a regular file."
else
  echo "$1 is a special file."
fi

if [[ -O $1 ]] ; then
  echo "You own this file."
else
  echo "You do not own this file."
fi
```

```
if [[ -r $1 ]] ; then
  echo "You have read permission on this file."
fi

if [[ -w $1 ]] ; then
  echo "You have write permission on this file."
fi

if [[ -x $1 ]] ; then
  echo "You have execute permission on this file."
fi
exit 0
------------------------------------------------------------    flist
#!/bin/ksh

# list files separated with name and date as header

ECHO=/unige/gnu/bin/echo

narg=$#
if test $# -eq 0
then
  $ECHO "No file requested for listing"
  exit
fi

if test $# -eq 2
then
  head=$1
  shift
fi

$ECHO `date`
for i in $* ; do
  $ECHO ' '
  $ECHO -n '------------------------------------------------------------    '
  if test $narg -ne -1
  then head=$i
  fi
  $ECHO $head
  cat $i
done
$ECHO ' '
$ECHO '------------------------------------------------------------    end'
```

```
exit 0
-------------------------------------------------------    grep2
#!/bin/ksh

# search for two words in a file

filename=$1
word1=$2
word2=$3
if grep -q $word1 $filename && grep -q $word2 $filename
then
  echo "'$word1' and '$word2' arre both in file:  $filename."
fi
exit 0
-------------------------------------------------------    guess1
#!/bin/ksh

# simple number guessing program

trap 'echo Thank you for playing !' EXIT

magicnum=$(($RANDOM%10+1))

echo 'Guess a number between 1 and 10 : '

while read guess'?number> '; do
   sleep 1
   if (( $guess == $magicnum )) ; then
     echo 'Right !!!'
     exit
   fi
   echo 'Wrong !!!'
done
exit 0
-------------------------------------------------------    guess2
#!/bin/ksh

# an other number guessing program

magicnum=$(($RANDOM%100+1))

echo 'Guess a number between 1 and 100 :'

while read guess'?number > '; do
```

```
  if (( $guess == $magicnum )); then
    echo 'Right !!!'
    exit
  fi
  if (( $guess < $magicnum )); then
    echo 'Too low !'
  else
    echo 'Too high !'
  fi
done
exit 0
-------------------------------------------------------------    minutes
#!/bin/bash
# count to 1 minute

i=0
date
while test $i -le 60; do
  case  $(($i%10)) in
    0 ) j=$(($i/10))
        echo -n "$j" ;;
    5 ) echo -n '+' ;;
    * ) echo -n '.' ;;
  esac
  sleep 1
  let i=i+1
done
echo
date
-------------------------------------------------------------    term1
#!/bin/bash
# setting terminal using select

PS3='terminal? '
oldterm=$TERM
select term in vt100 vt102 vt220 xterm dtterm ; do
  if [[ -n $term ]]; then
    TERM=$term
    echo TERM was $oldterm, is now $TERM
    break
  else
    echo '***** Invalid !!!'
  fi
done
-------------------------------------------------------------    term2
```

```
#!/bin/bash
# set terminal using select and case

PS3='terminal? '
oldterm=$TERM
select term in 'DEC vt100' 'DEC vt220' xterm dtterm; do
  case $REPLY in
    1 ) TERM=vt100 ;;
    2 ) TERM=vt220 ;;
    3 ) TERM=xterm ;;
    4 ) TERM=dtterm ;;
    * ) echo '***** Invalid !' ;;
  esac
  if [[ -n $term ]]; then
    echo TERM is now $TERM
    break
  fi
done
------------------------------------------------------------    tolower
#!/bin/bash
# convert file names to lower case

for filename in "$@" ; do
   typeset -l newfile=$filename
   eval mv $filename $newfile
done
------------------------------------------------------------    toupper
#!/bin/ksh
# convert file names to upper case

for filename in "$@" ; do
   typeset -u newfile=$filename
   echo $filename $newfile
   eval mv $filename $newfile
done
------------------------------------------------------------    end
```

Index