Introduction to shell programming using bash
Part I

Deniz Savas and Michael Griffiths

2005-2011

Corporate Information and Computing Services
The University of Sheffield

Email  M.Griffiths@sheffield.ac.uk
       D.Savas@sheffield.ac.uk
Presentation Outline

- Introduction
- Why use shell programs
- Basics of shell programming
- Using variables and parameters
- User Input during shell script execution
- Arithmetical operations on shell variables
- Aliases
- Debugging shell scripts
- References
Introduction

• What is ‘shell’?
• Why write shell programs?
• Types of shell
What is ‘shell’?

- Provides an Interface to the UNIX Operating System
- It is a command interpreter
  - Built on top of the kernel
  - Enables users to run services provided by the UNIX OS
- In its simplest form a series of commands in a file is a shell program that saves having to retype commands to perform common tasks.
- Shell provides a secure interface between the user and the ‘kernel’ of the operating system.
Why write shell programs?

- Run tasks customised for different systems. Variety of environment variables such as the operating system version and type can be detected within a script and necessary action taken to enable correct operation of a program.
- Create the primary user interface for a variety of programming tasks. For example- to start up a package with a selection of options.
- Write programs for controlling the routinely performed jobs run on a system. For example- to take backups when the system is idle.
- Write job scripts for submission to a job-scheduler such as the sun-grid-engine. For example- to run your own programs in batch mode.
Types of Unix shells

- sh Bourne Shell (Original Shell) (*Steven Bourne of AT&T*)
- csh C-Shell (C-like Syntax) (*Bill Joy of Univ. of California*)
- ksh Korn-Shell (Bourne+some C-shell) (*David Korn of AT&T*)
- tcsh Turbo C-Shell (More User Friendly C-Shell).
- bash Bourne Again Shell (*GNU Improved Bourne Shell*)

You can switch from one shell to another by just typing the name of the shell. `exit` return you back to previous shell.

Default shell on *iceberg* is bash. This is the most recently developed and best shell to use as it combines all the nice features of the previous shells.
Which “shell” I use?

- **sh** (Bourne shell) was considered better for programming.
- **csh** (C-Shell) was considered better for interactive work.
- **tcsh** and **korn** were improvements on c-shell and bourne shell respectively.
- **bash** is largely compatible with sh and also has many of the nice features of the other shells.

We recommend that you use bash for writing new shell scripts but learn csh to understand existing scripts.
Your Working shell

- The default shell on iceberg is bash.
- If you prefer, you can work with another shell.
- You can load the shell of your choice by simply typing the name of the shell. e.g. csh will start a c-shell.
- exit will revert you back to the previous shell (or simply log you out if you are already back to the initial log-in level).
- Each user’s default shell is determined by the system administrator via the /etc/passwd file.
- NOTE: On iceberg sh maps onto bash and csh maps onto tcsh
Shell Script Features

A shell script is a file containing a list of shell commands that will be executed when the file is invoked as a command.

- The first line of a script-file indicates the type of shell that script is written in. This is known as the “BANG !” line. It starts with the character pair `#!/` in columns 1 and 2.
- It is usually one of `#!/bin/sh`, `#!/bin/bash`, `#!/bin/csh` or `#!/bin/tcsh` corresponding to the shell of choice. Closer inspection will show that it simply points to the location of the shell in the filestore hierarch.
- When a script file is executed
  - the operating system simply loads the shell indicated by the bang-line,
  - passes it control
  - the loaded shell reads and executes the rest of file as input.
- If the “bang-line” is missing the default shell is selected.
Shell Script Features ... continued

• Excluding the “bang-line”, all lines starting with # are interpreted as comments.
• We recommend that you use comments (lines starting with #) in your shell script files to make them understandable.
• Once a shell-script file is written, you must give it execute-rights to be able to use it.
• This is done by using the chmod command:
  - `chmod u+x scriptfilename`
• You can now execute the shell-script:
  - `.scriptfilename`
Exercises: Initialisation of your shell environment
the <dot>.files in your directory

• Log into iceberg and investigate your file store by typing `ls` and then `ls -a`. What is the difference in output?
• Any filename starting with <.> is normally hidden.
• By convention these .files are utilised by the shell and by other programs for initialisation and systems settings purposes. In MS-windows terminology they are a bit like .ini files.
• There are three important <dot> files that are used by bash. These are: .bashrc, .bash_profile, .bash_logout
There is also a file called .bash_history that keeps a record of your previous commands.
• Investigate the contents of these four files if they exist.
• Now investigate the contents of the file /etc/bashrc. This is the first system-wide bash script we have encountered so far.
Logging into linux

- Login command will authenticate the user
- Login command will set the default shell by looking into file /etc/passwd
- Login command will set the correct environment and start the shell program.
- If the shell is bash it will execute the system-wide bash-profile file /etc/profile and the system-wide bash startup file /etc/bashrc
- It will then execute .bash_profile from the users home directory.
- The convention is to invoke .bashrc from the .bash_profile script.

Starting a shell

When a new bash shell is started, ‘this also happens automatically when a new xterm is started’ only the file .bashrc is executed unless the user specifies it as a login shell ( e.g., bash -l ) in which case /etc/profile, /etc/bashrc and .bash_profile are also executed.
Pre-defined environment variables

Exercises

• During an interactive bash shell type the command `set` and observe the output from it.
• Note that a similar ‘but not the same’ output can be obtained by using the `printenv` command.
• Every line of the set command output will list a variable and its current value. These are various mainly system variables already defined for you and are accessible from within the shell scripts you are going to write.
• You can display the value of a specific variable by typing `echo $variable_name`.
  For example: `echo $SHELL`, `echo $PATH`
Exercises

The obligatory Hello world !!!

#!/bin/bash
#
# The first example of a shell script
echo 'Hello World'

All lines starting with # are comments. Exception to the rule is that the first line (the bang! line) which tells the command interpreter which shell to use when interpreting the commands in the script. In this case the shell is loaded from /bin/bash.
Hello script exercise continued...

• The following script asks the user to enter his name and displays a personalised hello.

```bash
#!/bin/bash
echo "Who am I talking to?"
read user_name
echo "Hello $user_name"
```

• Try replacing “ “ with ‘ ‘ above to see what happens.

• Study the three scripts hello ,hello2 and hello3
Commenting

• Lines starting with # are comments except the very first line where #! Indicates the location of the shell that will be run to execute the script.
• On any line characters following an unquoted # are considered to be comments and ignored.
• Comments are used to;
  - Identify who wrote it and when
  - Identify input variables
  - Make code easy to read
  - Explain complex code sections
  - Version control tracking
  - Record modifications
User Input During Shell Script Execution

• As shown on the hello script input from the standard input location is done via the read command.

• Example

  ```bash
  echo "Please enter three filenames:"
  read filea fileb filec
  echo "The following files will be used: $filea $fileb $filec"
  ```

• Each read statement reads an entire line. In the above example if there are less than 3 items in the response the trailing variables will be set to blank ‘ ‘.
Few notes on the echo command

Echo command is well appreciated when trying to debug scripts.
Syntax: echo {options} string
Options: -e : expand \ (back-slash ) special characters
-n : do not output a new-line at the end.
String can be a “weakly quoted” or a ‘strongly quoted’ string. In the weakly quoted strings the references to variables are replaced by the value of those variables before the output.
As well as the variables some special backslash\_escaped symbols are expanded during the output. If such expansions are required the –e option must be used.
Echo command

- Special back-slash escape characters used with it:
  \n  newline
  \r  carriage return
  \t  tab
  \c  suppress trailing new line
  \f  form-feed
  \b  back-space
  \v  vertical tab

Example:
```bash
  echo -e "Hello $friend \t enter passcode:\c"
```
Quote characters

There are three different quote characters with different behaviour. These are:

“ : double quote. If a string is enclosed in “ s the references to variables (i.e $variable ) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the the string.

Example: echo “Today’s date is:” `date`
Environment variables

• Global environment variables
  
  When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.
  
  However, care is needed when you change the values of some of these variables, such as the PATH variable.

• Local environment variables
  
  Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell ‘i.e. is local to that shell’. When the shell execution finishes and the control returns back to the invoking level these variables are automatically destroyed.
global variables

• By convention global variables are given names in CAPITAL letters.
  ( Type set to get a list of all variables)
• When a new shell is created copy of all the global environment variables are taken and transferred to the new shell.
• Therefore the global variables are available to all subsequently created shells
• The following slide lists a few of the important global variables.
variables

• **Global environment variables**
  When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts. However, care is needed when you change the values of some of these variables, such as the PATH variable. Use `env` command to get a list of current global variables.

• **Local environment variables**
  Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell ‘i.e. is local to that shell’. When the shell execution finishes and the control returns back to the invoking level these variables are automatically destroyed.
global variables

• By convention global variables are given names in CAPITAL letters.
  (Type set to get a list of all variables)
• When a new shell is created copy of all the environment variables are taken and transferred to the new shell.
• Therefore the global variables are available to all subsequently created shells
• The following slide lists a few of the important global variables.
A few global (environment) variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHELL</td>
<td>Current shell</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>Used by X-Windows system to identify the display</td>
</tr>
<tr>
<td>HOME</td>
<td>Fully qualified name of your login directory</td>
</tr>
<tr>
<td>PATH</td>
<td>Search path for commands</td>
</tr>
<tr>
<td>MANPATH</td>
<td>Search path for &lt;man&gt; pages</td>
</tr>
<tr>
<td>PS1 &amp; PS2</td>
<td>Primary and Secondary prompt strings</td>
</tr>
<tr>
<td>USER</td>
<td>Your login name</td>
</tr>
<tr>
<td>TERM</td>
<td>terminal type</td>
</tr>
<tr>
<td>PWD</td>
<td>Current working directory</td>
</tr>
</tbody>
</table>
Defining Variables

- As in any other programming language, variables can be defined and used in shell scripts.
- Unlike in other programming languages variables are not typed.
- Examples:
  - `a=1234` # `a` is integer
  - `b=$a+1` # will not perform arithmetic but be the string ‘1234+1’
  - `let "b=$a+1"` will perform arithmetic so `b` is 1235 now.
  - Note: +,-,/,*,**, % operators are available. However `(( ))` syntax ‘see later’ is better and preferable to this form of arithmetic.
  - `b=abcde` # `b` is string
  - `b=‘abcde’` # same as above but much safer.
  - `b=abc def` # will not work unless ‘quoted’
  - `b=‘abc def’` # i.e. this will work.

**IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE “=”**
Referencing Variables

- Having defined a variable, its contents can be referenced by the $ symbol. E.g. ${variable} or simply $variable. When ambiguity exists $variable will not work. Use ${ } the rigorous form to be on the safe side.

- Example:
  a='abc'
  b=${a}def # this would not have worked without the{ }
  #enclosing a and would have tried to access
  # a variable named adef
Referencing Variables and Substrings

Contents of the variable can be referenced with the `${variable}`, `$variable` or `"$variable"` syntax.

A sub-string syntax introduced into bash allows access to a sub-section of a variables string-value.

`${string:position}` extracts a substring from string starting at `position`.

`${string:position:length}` extracts length number of characters starting at `position`.

Note that the first character is position=0 (and not 1!)

Example:

`a=‘abcdef’`

Therefore `${a}` contains ‘abcdef’

and `${a:3}` contains ‘def’ and `${a:3:1}` contains just ‘d’.

This avoids some the use of the `cut` command.
Sub-string Replacement

Syntax: \${variable/string1/string2/}

Example: If variable a contains ‘abcdefg’ then
\${a/fg/zz/} evaluates to ‘abcdezz’

The above syntax replaces the first occurrence of /string1/. If the aim is to replace all occurrences then

The syntax is \${variable//string1/string2}

Example: a=‘1230456099900’
\$a{//0/-} evaluates to ‘123-456-999--’
Default Values, indirect expansions

- Default Values: This subtle bash feature can be very useful in complex scripts where it is not always known that a local variable has been created.

  Example:

  ```
  infile=${infile:='data1'}
  ```

  The variable `infile` will be created if it did not exist and it will also be assigned the default value of `'data1'`

- Indirect expansions: A wild-card form of accessing variables is made possible.

  Example:

  ```
  echo ${!user*}
  ```

  All variables names ‘not values’ which start with `user` will be listed.
Deleting Unwanted Variables & Exporting Variables to child shells

• Use the `unset` command to remove a variable.
  e.g.    unset myvar1

• Assigning null to a variable clears the contents of that variable but does not remove the variable itself.
  e.g.  Myvar1=

• Having defined a variable its value can only be transferred to the child processes and shells by means of the `export` command.
  e.g.  myvar1=‘deniz’ ; export myvar1
  or            export myvar1=‘deniz’
Exporting Local Variables to child shells

- Child processes of the current shell will only be aware of the local variables if it is exported.
- However, changing the value of such exported variables in the child shell will have no effect in the parent shell.

Example;

```bash
export myvar1="deniz"
echo $myvar1   ( will print: deniz )
bash
  echo $myvar1   ( will print: deniz )
  export myvar1="savas"
  echo $myvar1   ( will print: savas )
exit
  echo $myvar1   ( will print: deniz )
```

Note: changing GLOBAL variables in subshells do not effect parent shell either.
**Starting shells**

Login shell contains all global variables + any local vars created by you while on login level.

Running a script or program starts a new shell which is the child of the current shell. All global variables are copied to the new ‘child’ shell but only the ‘exported’ local variables transferred to the new ‘child’ shell

The global variable SHLVL contains the hierarchical level of the current shell.
Source statement

- Sometimes you do not want to start a new shell but overlay the current shell with a new one.
- Invoking a script file as `source script` overlays the shell with a new shell and runs the script.

Exercise:
- define a local variable called myname
  e.g. `mynname='paris'
- run the script named `vexample1` as
  (1) `./localvars5`
  (2) `source localvars5`
  (3) `. localvars`
- check the value of local variable you created and also the ones that are created in `localvars5`.

Experiment with the `export` command.
Positional Parameters

• When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.
• $0 This variable that contains the name of the script
• $1, $2, ..... $n 1^{st}, 2^{nd} 3^{rd} command line parameter
• $# Number of command line parameters

Example:
Invoke : ./myscript6 one two buckle my shoe
During the execution of myscript variables $1 $2 $3 $4 and $5 will contain the values ;
one , two , buckle , my , shoe respectively.
Special Parameters

• As well as the usual positional parameters $1, $2, so on, there are a set of other special parameters that get created during the invocation of a script. Some of these are listed below;

  $*$  all the parameters.
  $0$  the name of the script that is running
  $#  total number of parameters to the script
  $$  process ID of the shell
  $@$  same as $* but as a list one at a time (see for loops later)
  $?  Return code ‘exit code’ of the last command

shift command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1, and so on ... It is a useful command for processing the input parameters one at a time.
Expansions

• Brace expansions: Generate multiple combination values
  
  \text{string1\{sta,stb,stc,std\}string2}

• Tilde expansion: substitute $\text{HOME}$ or home directory
  
  e.g. \text{cd ~/bin}

• Command output substitution:
  
  e.g. \text{today=`date`}
Arithmetic Operations in shell scripts

- Integer Arithmetic
  - `$(arithmetic expression)` allows us to perform integer arithmetic operations. This should be preferred to the older `exp` syntax for the Bourne shell.

Example:
```
a=$((4+3)) , b=$((a * 7)) , c=$((b / 3 - 1))
```

Note: spaces within `((   ))` do not matter so use them for readability.

Some of the allowed operations are given in the following table.
## Arithmetic Operations in shell scripts

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>var++ , var-- , ++var , --var</td>
<td>post/pre increment/decrement</td>
</tr>
<tr>
<td>+ , -</td>
<td>add subtract</td>
</tr>
<tr>
<td>* , / , %</td>
<td>multiply/divide, remainder</td>
</tr>
<tr>
<td>**</td>
<td>power of</td>
</tr>
<tr>
<td>! , ~</td>
<td>logical/bitwise negation</td>
</tr>
<tr>
<td>&amp; ,</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp; ,</td>
<td></td>
</tr>
</tbody>
</table>
Floating point arithmetic

- There are no built-in rules for performing floating point maths using shell variables but we can get around this problem by a subtle use of the `bc` basic-calculator command.

Example: add the contents of two variables (a & b) that contain decimal notation number strings and store the results in another variable (c)

```
a=3.412
b=7.438
c=`echo $a+$b | bc -l`
```
A few other useful tips

• Generating random numbers
Reference to the built in variable $RANDOM will return an integer in the range 0 to 32767.
Use an expression like the one below to get a random number that is between 0 and n-1.
   n=36; randnum=$(( $RANDOM % $n ))

• Getting a sequence of numbers
Use the seq command to generate a list of numbers.
For example;
   list=`seq 20`
will assign 1 2 3 .... 20 to the variable named list.
aliases

• Aliasing facility allows the opportunity to personalise the command environment, allowing the user to give more memorable names to some Linux commands or to define/change the default options of commands.

Examples:
alias dir=‘ls’
alias xterm=‘xterm -fn 8x15’
alias l.=`ls -d .*’

• The first word of an interactive command is checked if it is an alias and if so the value of the alias replaces its name.

• Aliases are only valid in the shell where they are defined. Therefore it is advisable to put all your alias definitions into to .bashrc file which gets run for every shell.

• Aliases are only expanded in the interactive shells.

• Remove aliasing by the unalias command;
  e.g. unalias dir
Debugging your shell scripts

- Generous use of the `echo` command will help.
- Run script with the `-x` parameter.
  
  E.g. `bash -x ./myscript`
  
  or `set -o xtrace` before running the script.

- For longer scripts, debugging can be applied to selected sections that are defined by the `set -x` and `set +x` commands.

Example:

```
set -x
commands in
the section to be debuged
:
set +x
rest of the script
```
Debugging shell scripts ...

• Verbose mode `-v` will print the shell line as it is about to be executed.
  e.g. `bash -v myscript`

• The `-x` and `-v` options can be combined.
  e.g. `bash -xv myscript`
  - Also set `-v` / set `+v` can be used in a similar manner to
    set `+-x` described in the previous slide.

• These options can be added to the first line of the script where the shell is defined.
  e.g. `#!/bin/bash -xv`

FURTHER STUDY IDEA: Type `set -o` to see further options. Check references to find out what they mean.
Keeping a record

- Script command can be used to make a record of everything that appears on the terminal.

  `script filename`

  where `filename` is optional and if not specified it is assumed to be `typescript`.

  Once started all terminal output will also be copied onto the scriptfile until a `^D` is received.

  **Note:** `set -o ignoreeof` can be used to avoid logging out accidentally by pressing `^D` too many times.
References

See the ‘bash’ sections of the following page:

http://www.shef.ac.uk/wrgrid/documents/links.html