The Shelx Tuesday Seminars:

Shell Scripting with the Bash

Tim Grüne

January 11^{th}, 2005

History of UNIX shells

Year	Shell	Author	Properties
1974	`sh`	Steven Bourne, Bell Labs	first shell; no history, no command line editing
1978	`csh`	Bill Joy, Berkeley	history, “built-ins”; C–language expression syntax
1983	`ksh`	David Korn	superset of `sh` with features from `csh`
$\approx$ 1988	`bash`	Chat Ramey *et al.*	The “Bourne Again SHell”: much extended `sh`
1990	`zsh`	Paul Falstad	combines features of `bash`, `ksh`, and `tcsh`. Has the reputation of being *too* configurable and powerful.

Most shells resemble the original Bourne `sh` syntax, apart from `csh` and its descendant `tcsh`. This means, a script compliant with `sh` can be run with `bash`, `ksh`, or `zsh`, but not with `(t)csh`. Therefore the use of `csh` is deprecated.

Bash is licensed under GPL, i.e. it is supported and distributed by the Free Software Foundation. This is one of the reasons why `bash` is popular among Linux distributions.

What is a shell? (1/3)

A shell is a command line interpreter: it allows the user to enter commands at a command prompt and process its results. It provides facilities to greatly enhance working with common UNIX utility programs.

Example: Interpretation of the asterisk (*)

> A directory contains three PNG-files: `image1.png`, `image2.png`, and `image3.png`. The command
>
> ```
> display *.png
> ```
>
> is expanded by the shell to
>
> ```
> display image1.png image2.png image3.png
> ```

This is an example of filename expansion. Without a shell one would have to type each filename explicitly.

What is a shell? (2/3)

A shell also allows to connect the results of different programs. This is done by pipelines and redirection.

This is useful because the UNIX world knows many small but helpful programs that are operated from the command line. Examples are

`cat`	write the file(s) contents to the terminal
`paste`	write lines of input files after each other
`grep`	find words/strings in a file
`sed`	replace strings line by line in a file
`awk`	more complex line by line editor

In addition to the commands installed on the system, shell also has built–in and user defined functions.

What is a shell? (3/3)

Modern shells can do much more than passing filenames to a command. They know basic arithmetics and conditional or looping constructs (`for/while`, `if ...`). They can be used as a scripting language.

A shell script is a sequence of commands written to a file. Writing a shell script is useful for tasks that are done more than once.

There are only a few differences between calling commands from a script or entering the command manually one by one. A script can accept command line arguments that influence its behaviour.

Invoking Bash (1/2)

Bash can start in three different ways: interactively as login shell, as non–login shell and non–interactively to run a shell script . The main difference between these is which files it reads for customisation.

login shell (started with `xterm -ls` or `bash --login`): Bash reads and executes commands from `/etc/profile` and from `~/.bash_profile` (or `~/.bash_login` or `~/.profile` if the previous ones do not exist).

At logout, Bash reads and executes commands from the file `~/.bash_logout`, if it exists ("~" expands to the user's home directory).

non–login shell A non–login is also connected to a terminal, i.e., it is interactive. Bash reads and executes commands from `~/.bashrc` unless the `--norc` option is given.

In order to avoid having to maintain two files, one can place the line

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

in the file `~/.bash_profile`. On SuSE Linux, this is the default setting.

Invoking Bash (1/2)

shell script When asked to run a shell script, Bash interprets the value of the variable `BASH_ENV` and reads and executes commands from that file. Normally this variable is not set, so nothing is done before execution of the script.

A script called `scriptname.sh` can be started either by

```
#> bash scriptname.sh
```

or by

```
#> ./scriptname.sh
```

For the latter to work, the script must be made executable (`chmod +x scriptname.sh`) and the first line of the script must read

```
#!/bin/bash
```

Variables

Variables can be used to store values and data. They are be set by (No spaces around the =–sign!)

```
#> NAME=value
```

or

```
#> export NAME=value
```

The latter version makes Bash pass NAME to programs started from the shell, otherwise they are only known to the current shell.

Variables are referenced by prepending a $–sign to the variable's `NAME`, e.g. `#> echo $NAME`.

A list of currently set variables can be obtained by the command `#> set`.

Environment variables are not only used by the shell. The shell only manages them. The `DISPLAY` variable is not used by the shell but all those that need to open a separate window.

Some useful Variables

`PATH`	colon separated list of directories of where to search for commands
`PS1`	primary command prompt, for interactive shell only. E.g.: `\u@\h:\w\$`
`$`	process ID of current process
`PWD`	name of current working directory
`OLDPWD`	name of working directory before the latest `cd` command
`PPID`	process ID of parent process
`MAILCHECK`	period for checking mail.
`HOME`	home directory of current user (`/xtal`, `/net/home/tg`). The command `#> cd $HOME` is equivalent to `#> cd`
`RANDOM`	returns a random integer between 0 and 32,767 each time it is called.

A few Bash variables are read-only and cannot be changed (e.g. `PPID`), but all others can be changed by the user. New variables can easily be introduced by simply defining them, e.g.

```
#> MY_OWN_VARIABLE="$UID $PWD"
#> echo $MY_OWN_VARIABLE
1000 /home/tim/Documents/tuesday_seminars/20050111
```

Command Line Editing

The original Bourne shell `sh` did not allow to modify a command. Once a character was typed, it could only be deleted by backspace.

Modern shells like the `bash` allow for much more sophisticated command line editing.

Moving the cursor	
`Ctrl-e`	move cursor to end of line
`Ctrl-a`	move cursor to beginning of line
Making corrections	
`Ctrl-d`	delete the character underneath
`Ctrl-k`	delete from cursor to end of line
`Ctrl-u`	delete from cursor to beginning of line
`Ctrl-_`	undo previous change(s)
`Ctrl-y`	paste previously deleted characters
`Ctrl-t`	swap character under cursor with its predecessor
`Ctrl-l`	clean up screen

Command History

Bash maintains a list of previously entered commands, the command history. When bash is not active, this list is kept in the file `~/.bash_history` (or the value of the variable `HISTFILE`). `$HISTSIZE` commands are kept on the stack, default is 500. The user can browse the history with the up and down arrow keys.

`Ctrl-r STRING` and then repeatedly `Ctrl-r` lets the user search backwards through the history for lines that contain `STRING`. It may contain spaces.

`!STRING` entered at the command line repeats the last command that started with `STRING`. A `:p` after `STRING` with no spaces places that command at the top of the history so that it can be recalled by hitting the up–arrow–key, without executing the command.

Redirection

A program that reads input from the command line is said to read form standard input. E.g., the program `ipmosflm` reads from the terminal a few parameters like detector type, filename–templates, etc.

One can also write this information into a file, say `mos.inp` and start mosflm with `#> ipmosflm < mos.inp` This has the same effect as typing the contents of `mos.inp` into the terminal.

If output is usual visible on the terminal, it is written to either standard output or standard error. If a program has a lot of output, one might want to catch it into a file.

Standard output is redirected by “1>” or simply “>”, standard error is redirected by “2>”. They can be used separately or combined:

```
#> mtzdmp data.mtz 1> data.log
#> mtzdmp data.mtz 2> error.log
#> mtzdmp data.mtz 1> data.log 2> error.log
```

To write both standard output and error to the same file, use the construct

```
#> mtzdmp data.mtz > data.log 2>&1
```

Pipelines

A pipeline is a sequence of commands separated by “|”. It can be used with programs that read their input from the command line and write to standard out.

```
#> grep "^ATOM" myfile.pdb | grep CA > mainchain.pdb
```

1. find lines beginning with `ATOM` in the file `myfile.pdb` and write them to standard output
2. redirect output to second `grep` which finds all lines containing “CA” and write them to standard output
3. redirect output to file `mainchain.pdb` which now contains the C_α–trace of the protein.

Aliases

An alias is a short–hand for a (simple) command.

Since an alias definition is not called recursively, a system command can be aliased with the same name, e.g.

```
#> alias ls='ls --color'
#> alias ll='ls -la'
```

The text within the single quotes on the right hand side is inserted whenever the string of the left hand side is entered as a command.

An alias can only be used in interactive shells, unless the `expand_alias` option is set (Bash options can be set with `#> shopt -s option_name` and unset with `#> shopt -u option_name`).

Functions

An alias is good for very short commands. For more complex commands, one can use a function. Functions can take command line arguments.

A function is defined by the optional word `function`, its name, followed by brackets and the function body enclosed by braces:

```
function ccp4i ()
   {
     if [ -z "CCP4"]; then
      source /xtal/xtal.setup-bash;
     fi
     ccp4i;
   }
```

This function tests whether the string referred to by the variable `CCP4` has zero length. If so, it sources the setup-file for crystallographic software (which sets up the ccp4 environment) and then executes `ccp4i`, which can now be found in the directory `/xtal/ccp4-5.0.2/ccp4i/bin`, which was added to the `PATH` variable by the ccp4 setup.

When a command is entered, bash first searches the aliases and functions before it tries to find the executable in one of the directories from the `PATH` variable and executes it.

Positional Parameters (1/2)

A script/program becomes more flexible if the user can pass arguments. A `bash` script accesses command line arguments through positional parameters.

Parameters are numberes `1...N` so that from within a script or function, they can be accessed with `${1}` through `${N}`.

The following are special parameters:

`$0`	name of the command that invoked the shell
`$*`	list of all parameters; "`$*`" (with double quotes) expands to single word "`$1 $2 ...`"
`$@`	list of all parameters; "`$@`" (with double quotes) expands to separate words "`$1`" "`$2`" `...`
`$#`	number `N` of positional parameters

Positional Parameters (2/2)

script name: ./03_positional.sh

```
#!/bin/bash

echo Process ID \$\$: $$
echo Value of \$0: $0
echo Number of arguments: \$\#: $#

echo Value of \$\*: $*
echo Value of \$\@: $@

echo Difference between \$\* and \$\@:
echo \"\$\*\" expands into one string:
for i in "$*";
do
        echo $i
done

echo \"\$\@\" expands into several strings:
for i in "$@"; do
        echo $i
done
```

```
#>./03_positional.sh one two three

 Process ID $$: 7182
 Value of $0: ./03_positional.sh
 Number of arguments: $#: 3

 Value of $*: one two three
 Value of $@: one two three

 Difference betwenn $* and $@:
 "$*" expands into one string:


 one two three


 "$@" expands into several strings:
 one
 two
 three
```

Arrays

Bash not only knows scalar variables, but can also handle arrays. They are initiated on demand, i.e.

```
#> my_array[5]="17*9="
#> let my_array[3]=17*9          # "let" lets bash calculate
```

are both possible without declaration or definition of array[0-2] and array[4].

When referring to array elements, braces must be used:

```
#> echo ${my_array[5]}${my_array[3]}
17*9=153
#> echo $my_array[5]
[5]
```

The second command means `echo ${my_array}[5]` which is equivalent to `echo ${my_array[0]}[5]`

Separating Commands and Command Lists

A shell script consists of a series of commands.

In Bash, commands a separated by semi–colon `";"` or a <newline>.

Several commands can be packed into one list by parentheses or braces:

```
( COMMAND1 ; COMMAND2 ; COMMAND3 ...)
```

The parentheses cause the commands to be executed in a subshell. This means that for example variable assignments have no effect after the end of the list.

```
{ COMMAND1 ; COMMAND2 ; COMMAND3 ...; }
```

With braces, the commands are executed in the current shell context.

NB: with the braces–construct, spaces around them and the final semi–colon are required.

Exit status of a list is the status of the last `COMMAND`.

Conditional Constructs (1/3)

Conditional and looping constructs are very important especially for scripting. Bash has the following syntax for `case` and `if`, the two basic constructs:

```
case WORD in
                PATTERN )  COMMAND(s) ;;
    ( PATTERN | PATTERN )  COMMAND(s) ;;
                 .
                 .
                 .
esac
```

More than one command per case must be separated by semi–colons (;). Only the first match is executed.

An example from our ccp4–setup:

```
case $HOSTNAME in
    node?)  export CCP4_SCR=/local/$USER ;;
    *)      export CCP4_SCR=/usr/tmp/$USER ;;
esac
```

Conditional Constructs (2/3)

```
if TEST-COMMANDS; then
    COMMAND(s);
elif OTHER-TEST-COMMANDS; then
    OTHER-COMMAND(s);
else
    ALTERNATE-COMMAND(s);
fi
```

The `elif` and `else` parts are optional.

The `TEST-COMMANDs` can be

a list of commands They are true if their return status is zero

an arithmetic expression The expression must be enclosed in `((...))`. It is true, if the return value is non–zero. Bash knows ordinary maths (+, -, /, *, ** (exponentiation), . . .), bitwise operators (&, |, . . .), logical operators (&&, ||, ?:), etc., also known to other programming languages. NB: Bash only knows arithmetics of integers (2/3=0).

Conditional Constructs (3/3)

a conditional (boolean) expression A boolean expression is enclosed by double square brackets `[[...]]`. Many expressions are available to test properties of files and strings.

`-a FILE`	true if file exists	`-w FILE`	true if FILE exists and is writable
`-d FILE`	true if file exists and is a directory	`-z STRING`	true if STRING has zero length
`-f FILE`	true if FILE exists and is a regular file	`ARG1 OP ARG2`	ARG1 and ARG2 are integers, OP one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`
`-h FILE`	true if FILE is a symbolic link	`STRING1 OP STRING2`	string comparison. OP can be `==`, `!=`, `<`, or `>`. Strings are ordered lexicographically.
`-s FILE`	true if FILE exists and has size greater than zero	`FILE1 -nt FILE2`	True if FILE1 is newer than FILE2, or if FILE1 exists and FILE2 does not

The expression can be negated by a preceding “!”

Looping Constructs (1/2)

Bash is aware of `while`, `until`, and `for` loops. The first two have the same syntax,

```
while/until TEST-COMMAND(s); do COMMAND(s); done
```

The `for`–loop has two alternate syntaxes:

```
for NAME in WORDS; do COMMAND(s); done
```

During each run of the loop, NAME takes the value of the next WORD and can be referred to by `${NAME}`. E.g.

```
#> for i in *.eps; do convert $i png:${i%eps}png ; done
```

uses the program `convert` to make copies in png–format of all eps–files in the current directory.

Looping Constructs (2/2)

The second form uses arithmetic expressions, similar to the syntax in C:

```
for (( EXPR1 ; EXPR2 ; EXPR3 )) ; do COMMANDS ; done
```

E.g.

```
 for (( i=35 ; i<45 ; i+=1 )) ; do
        (shelxe -h -m20 -s0.${i} data data_fa
         mv data.lst data_solvent_${i}.lst) &
        (shelxe -h -m20 -i -s0.${i} data data_fa
         mv data_i.lst data_i_solvent_${i}.lst)
done
```

runs shelxe and shelxe with inversed substructure and solvent content between 35% and 45% with 0.5% steps. After completion the log–files are given unique names to prevent overwriting.

Pattern Matching (1/2)

Bash understands patterns to match strings (like filenames).

`*` The asterisk matches any string, including the empty string

`rm *` deletes all files in the current directory

`rm name*.png` deletes all files that start with the word `name` and end with the word `.png`

`?` The question mark matches any single character.

`ls cryst_00?.img` lists the files `cryst_000.img` through `cryst_009.img`, but also `cryst_00a.img` or `cryst_00?.img`, etc., but only those that exist.

`[...]` Square brackets describe groups and ranges of characters

Pattern Matching (2/2)

Square brackets replace a single character:
`ls cryst_00[01].img` is equivalent to `ls cryst_000.img cryst_001.img`. Any number of characters can be placed in the square brackets and several square brackets can be used at the same time, as in
`ls cryst_[12]_00[01].img`

To save typing, ranges can be used as in `[a-d0-3]` = `[abcd0123]`

To match all but the enclosed characters, the first character must be `^` or `!`

If the `extglob` option is enabled (`shopt -s extglob`, not enabled by default), pattern lists can be used, separated by "`|`":

`?(PATTERN-LIST)` matches zero or one occurrence of the given patterns. `*(...)` matches zero or more, `+(...)` one or more, `@(...)` exactly one occurrence of the patterns.

Parameter Expansion (1/2)

In a previous example the expression `${i%eps}` was used to remove a trailing `eps` from the string stored in the variable `i`. This is an example of Parameter expansion.

```
${PARAMETER%WORD}
```

Where `WORD` is a pattern or pattern list. The pattern `WORD` is expanded and the shortest trailing match removed from `PARAMETER`. A second `%` removes the longest match.

```
#> W=image_1_0003.end
#> echo ${W%*([0-9]).end}
 image_1_0003
#> echo ${W%%*([0-9]).end}
 image_1_
```

A `#` instead of `%` (or two, respectively) does the same from the beginning of the value of `PARAMETER`.

Parameter Expansion (2/2)

Substitution (like in `vi`) is done with

```
${PARAMETER//PATTERN/STRING}
```

where one or two "`/`" after `PARAMETER` has the same meaning.

Substrings can be extracted with

```
${PARAMETER:OFFSET:LENGTH}
```

`OFFSET` is zero–based, i.e., the first character in `PARAMETER` has offset `0`. If `:LENGTH` is omitted, the value expands to the end of `PARAMETER`.

Further Reading

Most information of this talk stems from trial–and–error or while working with the Bash and from the reference manual, available as info–file, (`#> info bash`), or on–line at http://www.gnu.org/software/bash/manual/bashref.html.