united nations educational, scientific and cultural organization

international atomic energy agency

the abdus salam international centre for theoretical physics

40th anniversary 1964 2004

Third Workshop on

Distributed Laboratory Instrumentation Systems

22 November - 17 December 2004

Abdus Salam ICTP - Trieste, Italy

1594/8

Shell Programming

P. Bartholdi
Observatory of Geneva, Switzerland

strada costiera, 11 - 34014 trieste italy - tel. +39 040 2240111 fax +39 040 224163 - sci_info@ictp.trieste.it - www.ictp.trieste.it

Shell Programming

P. Bartholdi[†*]

[†] *Observatory of Geneva*
CH-1290 Sauverny – Switzerland

Lecture given at the:
Third Workshop on
Distributed Laboratory Instrumentation Systems
Trieste, 22 November — 18 December 2004

LNS

[*] Paul.Bartholdi@obs.unige.ch

Abstract

In this chapter we look at the use of Unix commands in general and at shell scripts as a very high level computer language, where the elementary units are not only numbers or strings, but also files, tasks, priorities etc.

The Unix commands most useful for shell programing are presented in more details.

The two families of Unix shells, Bourn and C shells, are presented, pinpointing where they differ and what are their respective advantages.

Many short examples illustrate the text, without to much theoretical considerations.

Keywords: Unix, Unix file system, Shell Programming, C-Shell, Bourn Shell.
PACS numbers:

Contents

1 UNIX Tools

The goal of this section is not to introduce UNIX *per se*, but to show how some UNIX tools can help in the production of good software.

1.1 UNIX as a Programming Language

Forty years ago, much programming was done in assembler, if not with wires. Then higher level languages like Fortran, C, Cobol etc. permitted the development of codes more or less independent of the hardware and operating system, that is much easier to read, that can be developed in reusable modules. Yet, the basic building blocks are still relatively low level instructions that are combined into higher and higher modules to form a single large program, where the modules are 'hard' interconnected.

The pipes and redirections, the very large number of simple standard tools available in UNIX and the facilities to build newer tools in the same spirit, and then interconnect them into streams and shells, make UNIX an ideal interactive programming environment.

1.2 Pipes and Redirections

Pipes permit to write small modules dedicated to simple tasks, and to interconnect them through standard input/output. Such modules are much simpler to develop and test individually, while the pipe checks for the interfaces. When fully tested, these modules can be put together in larger ones. The number of successive pipes is practically unlimited.

```
ls -l | sort +4n | less
```

Redirection is a good way to have all input data (including test ones) in files that can be text-edited. Output redirection, in particular using `tee`, builds sets of files against which future version's output can be compared (use `diff` for that).

```
prog < test.data > test.results
```

or

```
prog < test.data | tee test.results | less
```

While every module has only one *standard input*, it has effectively two output ones: the *standard output* and the *standard error output*. The output redirection sign (>) followed by an ampersend (&) merge the two output strams into the given file. Similarely, the pipe sign (`|`) followed by an ampersend merge the two output streams into a single one as input to the following programme. Finaly, if the variable `noclobber` is defined, then the redirections will not overwrite old existing files, except if the redirection operator is followed with a `!` sign,

Here is a summary of all redirections and pipes (tcsh):

`<`	file	redirect *file* into standard input
`<<`	”Word”	redirect the following lines, until the one starting with ”Word”
`>`	file	redirect standard output to *file*
`>>`	file	append standard output to the end of *file*
`>&`	file	redirect standard output and standard error to *file*
`>>&`	file	append standard output and standard error to the end of *file*
`>!`	file	same as above, but ignore *noclobber*
`>&!`	file	same as above, but ignore *noclobber*

1.3 Five ways to input data into a programme

a) type data by hand
b) `prog < data-file`
c) `cat data-file | prog`
d) `echo "data" | prog`
e)
```
cat << FOF | prog
...
   data
...
FDF
```

1.4 Output Redirection in `ksh`

In `ksh`, the standard output is considered descriptor 1 and the error output descriptor 2. `2>&1` duplicates the error output on the standard output. To get both onto a file, the following form should be used (order is important):
`...  1>` *file* `2>&1`

1.5 Aliases and functions

Every complex command that may be used regularly could be aliased into a simple mnemonic name :

`alias mnemonic='`*`equivalent string`*`'`

The exact form of *alias* depends on the shell used. Here I have adopted the *bash* form.

Many examples of aliases are given below.

Aliasing into usual UNIX command should be carefully avoided if the use of the original version can be dangerous when the aliased one is expected.

`alias rm='/bin/rm -i'`

is a typical example. In another environment, `rm` will not ask you for confirmation when you expected it.

Inversely, tools that require a mode, should specify so: use "`~/bin/rm -f`" and not "`rm`".

But:

```
alias ls='/bin/ls -CF'
```

is a perfectly acceptable one. If it is not yet the case, put this alias in your `.tcshrc` or `.bashrc` file. `ls` will then automaticaly append a "`*`" at the end of executable files, a "`/`" at the end of directories and a "`@`" at the end of symbolics links.

Notice that, except in *csh* and *tcsh*, it is not possible to pass parameters to an alias. In most cases, but not with *csh* or *tcsh*, a function can replace an alias. In *ksh* and *bash*, a function is defined by:

```
function name() { commands }
```

Inside the function body (*commands*), parameters are referred by their positions in the calling statement, that is `$1` for the first parameter, and so on. See the examples bellow, page 3.

The keyword `function` itself is optional in *bash*.

Another method, probably safer than an alias and shell independent, is to have a reserved `~/bin` directory, and a corresponding scripts for each alias:

Put in your `.login` file a command:

```
PATH=~/bin:$PATH
```

and then:

```
echo "/bin/rm -i" > ~/bin/rmi
```

This will create a file, usually called a script (do not forget to make it executable), instead of the `alias` command. Typing `rmi` will execute that file.

1.6 Searching Tools

`grep` is a very powerful tool to do all sorts of searches and filters, in particular as part of a pipe stream. It looks for all occurrences of a pattern inside a set of files, and print the corresponding lines. It has many options (see the man pages), among them three avec very useful: `-h` suppress the prefixing of filenames on output, `-i` ignore case distinctions, and `-v` to select non-matching lines.

For example, finding all files that use `stdio.h`:

```
grep stdio.h *.c *.h  or        *.[ch]
```

Printing error messages only, with full output into a file:

```
test < test.data | tee test.res | egrep -i error
```

`grep` can also be used very effectively to "search" through a "data base". Suppose that you have a file with names, phone numbers and remarks, more or less in free form, another with hints on different subjects concerning your programs etc.

Then you can define the following aliases (*tcsh*) or functions (*bash, ksh*):

```
alias    help  "egrep -ih \!* ~/.help ./.help"
function help(){egrep -ih $1  ~/.help ./.help}
alias    tel   "egrep -ih \!* ~/.phones /share/phones"
function tel()={egrep -ih $1  ~/.phones /share/phones}
```

`help` *xxx* will print all lines from `~/.help` and `./.help` that contains the string "*xxx*".

`tel` *abc* will do the same for the phone files. With `tel` or `help` you can look for anything, not necessarily name or first name, but also for partial phone numbers etc. `tel 0039` will list all entries in Italy, while `tel rinus` will find our director's one.

Here is another application, to list only the files that have been modified this day in the current directory:

```
alias today 'set TODAY=`date +"%h %d"`; ls -al | egrep $TODAY'
```

A similar command to see all files modified this day, in alphabetical order:

```
alias Today 'find .  -ctime 0 -print | sort'
```

1.7 Looking for parts of a file

`head` and `tail` can be used to select only a few useful lines:

To see only the first line of a set of subroutines:

```
head -1 *.c
```

To see only the largest (or the most recently modified) files:

```
ls -l | sort +4n -5 | tail -16
ls -rtl | tail
```

Long output could also be piped into `more` (or `less`, `most`).

`uniq` can be combined efficiently with `sort` to find "*words*" that are rarely used, and so possibly wrong (`sort -u` would do the same).

1.8 Stream Editor: `sed` and `gawk`

`sed` is a very simple but powerful editor that can be inserted in the middle of a stream. `gawk` can be used in the same way for very complex text manipulation. The simplest using of `sed` looks like:

```
...  | sed -e 's/abc/efgh/g' | ...
```

It will simply replace everywhere the pattern "*abc*" with "*efgh*". The first character after `s` will be used as the separator, it is not necessarily a `/`.

Here is a more elaborated example:

```
#!/bin/csh
# add a new user (board), in group 501
# and set the same encrypted passwd as in other machines.

/usr/sbin/adduser -g501 board
set today=`date +'%Y%m%d:%H%M'`
cd /etc
cp passwd passwd_$today
sed -e 's/board\:\!/board\:7s0kry.rn.dco/' passwd_$today > passwd
```

`gawk` is a very complex program for which the manual is more than 300 pages, but it can also be used very effectively as a single line program. In the simplest case, `gawk` is used to reorder the *fields* or choose among the fields in every lines. For example:

```
awk '{ print $3 $7}' file
```

will leave only the third and seventh fields.

Here is little more complex example:

```
#!/bin/tcsh
# lock all users with no password
# put a ! in the second field of the file /etc/passwd
# if that field is empty.

cd /etc
set today=`date +'%Y%m%d:%H%M'`
cp passwd passwd_${today}
/bin/awk ' BEGIN {FS=":"; OFS=":"} \
      { if(NF>=7) { if($2=="") $2="!";
                    print $0 } } ' passwd_${today} > passwd
```

1.9 Character conversion using `tr`

`tr` is intended to do all sorts of character conversion, including special characters, and optionally to replace strings of the same character by a single one.

Normally, two strings of characters are given as parameters to `tr`. `tr` will replace all occurances of characters in the first string by the corresponding character in the second string.

If the option `-d` is given, then the characters from the first string are deleted.

If the option `-s` is given, strings of the same characters are replaced by a single one.

Special characters are represented with \ and a character.

`\a`	ctrl G	bell
`\f`	ctrl L	form feed
`\n`	ctrl J	new line
`\r`	ctrl M	carriage return
`\t`	ctrl I	tab
`\v`	ctrl K	vertical tab
`\`*nnn*	octal value	

It is also possible to give a *class* of characters instead of a string. In this case, the *string* should have the form `'[:`*class*`:]'` , where *class* is one of `alnum alpha digit cntrl blank lower upper punct`.

Here are a few examples of using `tr`. The first three are equivalent. The last two can be very usefull if you have a mix of PC, Mac and Unix machines.

```
cat myfiile.c | tr ABC...Z abcd...z > ...
cat myfiile.c | tr A-Z a-z > ...
cat myfiile.c | tr '[:upper:]' '[:lower:]' > ...
```

```
echo $PATH | tr ":" "\n"
tr '\r' ''   < dos_file > unix_file
tr '\r' '\n' < mac_file > unix_file
```

1.10 Use of the `history`

`tcsh` keeps a log of the last *n* commands. *n* is defined with the command `set history=`*n* in the file `.cshrcr` or `.tcshrc` .

This log can be used in the following ways:

`history`	prints (on screen) the list of the last *n* commands executed,
`fc`	repeats the last command,
`fc` *n*	repeats a given command,
`fc -`*n*	repeats a given relative command,
`fc` *abc*	repeats the last command starting with the same letters,
`fc -s/`*old*`/`*new*`/g`	repeats the last command with editing (substitution [+global]),

Part of the repeated command can be re-used by the following *word designators*:

0	word 0 (= command)
n	n^{th} word
^	first word
$	last word
m–n	words m through n
–	words 0 through but last
	words 1 through last
%	word matched by the string

The word designators can be modified by appending a modifier to the specifier: <*specifier*>:<*modifier*>

r	root of the file name
e	extension of the file name
h	head of the path (but last comp)
t	tail of the path
s/old/new/	substitution
g	global (comes before s)
p	print but no execution
q	quote words
u	make first lower case letter upper
l	make first upper letter case letter lower

1.11 Command/file name completion

After you have typed a few letters of a command or file name:

<*TAB*> will complete it if possible and unique,

<*ctrl*>d will list all possible completions.

1.11.1 Finding something in a large directory tree – `find`

`find` allows to search through any directory tree, looking for matching file names or files modified before or after a given date for example, and then execute any sort of command, like printing file name with full path, deleting, executing a `grep` on them etc.

`find` has many options, but we will see only four. Refer to the man pages for all other ones.

`find . -name` <*file_name, possibly with wild card*> `-print`

`find . -ctime` <*n*> `-exec` <*command*>

In the command, use {} to replace the file name, ending the command with `\;`

The first parameter ("`.`") is the starting point, root of the directory we are searching.

The second is the selection criteria, according to file names or times.

Then comes the execution for all files that match the selection criteria.

In facts, many selection criteria and many executions can be used simultaneously. Selection criteria can possibly be joined by `or` or `and` and `not`.

`find` without any execution part simply produces a list of the selected files on the standard output. This can be used with `cpio` to copy directories recursively. Examples:

1. Remove all core files, printing their full path:

   `find . -name core -exec rm -f {} \;`

2. List all files created today in any subdirectory:

   `find . -ctime 0 -print`

3. Search for use of `stdio.h` in all c files:

   `find . -name \*\.c -exec grep stdio {} \;`

4. copy a directory tree on an other place:

   `find` <*source directory*> `| cpio -dpm` <*destination directory*>

1.12 Executing just What is Necessary, using `make`

When a project gets larger, it becomes more and more difficult to track which compilations, link and execution are necessary.

`make` permits to do such operations automatically, based on declared dependencies and last modification time. The set of commands executed in each case is completely open and not restricted in any way to compilation or link. Further, the dependencies can be given explicitly, supplied by compilers like `gcc -M`, or even assumed implicitly by `make` itself in many cases from the file suffixes.

The use of implicit assumptions make it faster to write but more difficult to read the dependency file.

The general form of a dependency file (usually named `Makefile`) is the following:

```
target(s): dependencies
<TAB>      commands to produce the target(s)
```

`make` without a parameter will check the first target for dependencies, and then recursively through the file. If a target is older than a dependency, then the corresponding commands are executed.

If `make` is used with a parameter (a target in the `Makefile`), then the search starts from this target.

Here is a small example of a `Makefile`

```
all:     prog test

prog:    main.o  sub.o
         $(LINK.c) -o $@ main.o sub.o

main.o:  incl.h main.c
         gcc -c main.c

sub.o:   incl.h sub.c
         gcc -c sub.c

test:    prog test.data
         prog < test.data > test.results
```

`touch` can be used to change the date of last modification.

`make` can also be used as a simple user interface for commands, when there are dependencies among them. Suppose that you have a dBase on which you can edit, make extraction, preformat, visualize or print. The user could then say: `make visualize` or `make edit`, and all necessary operations will be done automatically. Here is the corresponding makefile:

```
all : catalogue stickers

catalogue : Catalogue.dvi
        dvips -Php0d Catalogue

stickers : Stickers.dvi
        dvips -Php0 Stickers

catalogue.win : Catalogue.dvi
        xdvi Catalogue &

stickers.win : Stickers.dvi
        xdvi Stickers &
```

```
Catalogue.ps : Catalogue.dvi
        dvips Catalogue -o

Stickers.ps : Stickers.dvi
        dvips Stickers -o

Catalogue.dvi : Catalogue.tex catalogue.tex
        latex Catalogue

Stickers.dvi : Stickers.tex stickers.tex
        latex Stickers

catalogue.tex : m.rdb
        report catalogue.report < m.rdb > catalogue.tex

stickers.tex : m.rdb
        report stickers.report < m.rdb > stickers.tex

m.rdb : mediatheque.rdb
        cp mediatheque.rdb m.rdb

mediatheque.rdb : mediatheque.db
        m.awk mediatheque

clear :
        rm catalogue.tex stickers.tex Catalogue.dvi Stickers.dvi \
        Catalogue.ps Stickers.ps Catalogue.log Stickers.log      \
        Catalogue.aux Stickers.aux
```

If the files reside on more than one machine (using NFS for example), they should all be synchronised with `ntp` or similar time protocols.

For very large projects, when many persons are involved in the development, `make` is not sufficient. `make` ignores the notion of version or file locking that are necessary in these circumstances.

Other tools exists for them, in particular `sccs`, `RCS` or `CV`. `diff` and `patch` can be used to keep track of incremental updates and versions (including the recovery of previous code).

1.13 RCS and SCCS: Automatic Revision Control

RCS and SCCS designate sets of tools that help maintaining revisions of a product. Only RCS will be discussed; SCCS offers approximately the same capabilities while having an older, clumsier syntax. CVS is intended for the simultaneous update of files by many users.

If a program of a certain importance is being developed, it is essential to keep *all* versions of the source code — not just the last, or the ten last. All versions should be numbered; a log file should account for all the modifications made between two numbers; version numbers should be allowed to ramify in a tree-like manner; the binary code produced should be stamped with the version number; and if many people work on the same project, there should be some coordinating means between them.

RCS is a set of tools for UNIX that manages automatically these tasks. Text files are normally hidden by RCS. A developer may *check a file out*, that is make it visible in his directory for modification, while locking other developer's access to it; edit it, write appropriate logging information; and *check it in back*. Initially, a file `f.c` is placed under RCS' supervision with

```
ci f.c
```

with initial version 1.1. The file is moved to a special directory, usually `~/RCS/`. An edit cycle would now be:

```
co f.c
edit f.c
ci f.c
```

If you have EMACS, you may use its built-in capabilities to simplify this process: edit the file using its true path (`~/RCS/f.c`), and type *Ctrl-X* and *Ctrl-Q* to check the file in and out respectively.

It is not necessary to modify your `Makefiles`, as `make` automatically checks out and deletes files it doesn't find. If you really wanted to, you would just put:

```
...
f.c: /home/mickeymouse/RCS/f.c
<TAB>   co $<
...
```

RCS can stamp source and object code with special identification strings. To obtain them, place the marker "`$Id$`" somewhere inside your source file. `co` will automatically replace it with `$Id:` *filename revision_number date time author state locker*`$` and the marker "`$Log$`" is replaced by the log messages that are requested during a check-in.

RCS keeps all your previous versions through *reverse deltas*, i.e. keeps the last version in full, and reverse diff's to obtain previous revisions. These are accessed through

```
co -r<revision #>
```

and a sub-branch, new level major release etc. may be defined with

```
ci -r<new revision #>
```

Besides `ci` and `co`, RCS provides a few commands:

`ident`	extracts identification markers
`rlog`	extracts log information about an RCS file
`rcs`	changes an RCS file's attribute
`rcsdiff`	compares revisions

Refer to the manual pages for more detail.

1.13.1 RCS in a multiuser environment

UNIX by itself provides no file lock, neither file access control. But all the nuts and bolts are present.

For a good multiuser system with personalized file access control,

- create a user `rcs`, without terminal access (no shell) and locked password (`*LK*` in `passwd` file),
- make `RCS` directories belonging to `rcs`,
- for each file, use `rcs -a` to give access to every authorized users.

1.13.2 Remarks concerning RCS

1. The directory `~/RCS` is **not** made automatically (use `mkdir RCS`)
2. `ci` will not move `...c,v` files automatically to `RCS` (use `mv`)
3. `co` and `ci` will look automatically in `~/RCS/` if the file is not found in the current directory, and `~/RCS` exists.
4. `co` and `ci` will **not** lock automatically the files, use `co -l` instead.
5. `co` and `ci` work also on wild card. For example, `co -l *.c` will extract all `.c` files at once.
6. `rcs -l` *file* will lock the file. This is necessary if you modified a non locked file.
7. `rcs -U` / `rcs -L` *file* will enable/disable the file, doing strict locking.

1.13.3 Revision Control for Group Programming

As stated above, RCS is totally insufficient when many programmers work simultaneously on a project. CVS was developed to help with multiple revision trees and version merging. It includes down– and up–load from remote hosts. ARCH is a modern extension of CVS.

2 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 constructions, as variables, conditionals and loops, input and output, even some rudimentary arithmetic. Shell programming cannot replace C programming, in particular it is much slower, but it can be very effective to organize together the repetitive and possibly conditional execution of programs.

Writing script files can have two other advantages:

– They can be edited until it works, 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. 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 telling the system which one 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).

2.1 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*.

2.2 Quotes

Two quotes symbols can be used: `'` and `"`.

Inside `' '`, no special character is interpreted.

Inside `" "`, then $, `` ` ``, `!`, and \ are the only ones 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
```

2.3 Parameter passing

A command can be followed by parameters as “words” separated with 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 as a parameter.

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

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

2.4 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 space around the = sign, or read from the terminal or a file.

```
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"
```

2.5 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 the variables `LD_LIBRARY_PATH`, `MANPATH` and `PATH`.

The directory names are separated with 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, 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 automaticaly 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 PD program `envv` :

  `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*.

2.6 Reading data

Variables can be read from the keyboard with the `read` command as seen above. 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
```

2.7 Loop – `for` command

In `bash` and `ksh`, the command `for` permits to loop over many commands with a variable taking successive values from a list (See section 2.9 for the `csh` (`foreach`) equivalent).

The syntax is:
`for` <*variable name*> `in` <*list of values*> `; do`)
<*commands*>
<*commands*>
`...`
`done`

Here are a few examples using `foreach` in `csh` scripts. Try to rewrite them in `ksh` ones.

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.

2.8 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, look in the man pages of `csh` for a complete list.
Example: Save all executables and recompile:

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

2.9 bash/ksh and csh command syntax compared

Today, many people use `tcsh` for interactive work. Other prefer `bash` or `ksh`. It has so many goodies. But for shell programming, writing scripts, the choice is really open between `sh` and its offspring (`ksh`, `bash...`) on one side, and `csh` on the other. `ksh` or `bash` are now the default standard on Linux, probably the simpler yet most powerful of all. `csh` on the other end has the advantage of being a subset of `tsch`, with which the user is probably more comfortable. As with many other choices with computers, it has become a question of religion. Make your mind!

If your problem is more complex, if you need arrays, if you manipulate many files, then probably neither `bash` or `csh` are sufficient.

`awk` is almost ideal 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 whish, 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 `tsch`, it is not part of the system and has to be installed specifically by the "system manager".

The following pages compare the main commands used in `ksh` and `csh`. As you will see, some are missing on one or the other side, others are definitely simpler on one side, and many are quite similar.

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

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`

	ksh	csh

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

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

2.10 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` can not be caught or ignored, and will bring your shell to an end. `STOP` and `CONT` allows to stop temporarely a shell (or any task) and then restart it without loosing 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
#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)
```

2.11 Sample shell scripts

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

Here is a table of commands and corresponding scripts where they are used. The scripts bellow are in alphabetical order. Their names appear in the listing at the right, after a long dash line separating the various scripts. They are written in `ksh` or `bash`, but are easily converted to `csh`.

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

```
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 ˆC

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=1
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
```

2.12 Use of the `history`

`tcsh` keeps a log of the last *n* commands. *n* is defined with the command `set history=`*n* in the file `.cshrcr` or `.tcshrc` .

This log can be used in the following ways:

`history`	prints (on screen) the list of the last *n* commands executed,
`fc`	repeats the last command,
`fc` *n*	repeats a given command,
`fc` *-n*	repeats a given relative command,
`fc` *abc*	repeats the last command starting with the same letters,
`fc -s/`*old*`/`*new*`/g`	repeats the last command with editing (substitution [+global]),

Part of the repeated command can be re-used by the following *word designators*:

0	word 0 (= command)
n	n^{th} word
ˆ	first word
$	last word
m–n	words m through n
–	words 0 through but last
	words 1 through last
%	word matched by the string

The word designators can be modified by appending a modifier to the specifier: <*specifier*>:<*modifier*>

r	root of the file name
e	extension of the file name
h	head of the path (but last comp)
t	tail of the path
s/old/new/	substitution
g	global (comes before s)
p	print but no execution
q	quote words
u	make first lower case letter upper
l	make first upper letter case letter lower

2.13 Command/file name completion

After you have typed a few letters of a command or file name:
<*TAB*> will complete it if possible and unique,
<*ctrl*>d will list all possible completions.

3 References and Bibliography

The following bibliography is not necessarily very coherent. It contains old and new books, as well as some reference articles. They are all in my personal library. I have not read all of them, but they all contain something that impressed me and changed my way of using computers.

Many of these books have been reprinted, some re-edited, and the dates given may not be up-to-date.

3.1 Unix Tools and Shell Programming

- **Bolinger D. and Bronson T.** Applying RCS and SCCS. *O'Reilly & Associates 1995*

- **DuBois Paul** Type Less, Accomplish More Using csh & tcsh. *O'Reilly & Associates 1995*

- **Kernighan B.W. and Plauger P.J.** Software Tools. *Addison-Wesley 1976*

- **Kochan S. amd Wood P.** Unix Shell Programming. *Sams 2003*

- **Miller W.** A Software Tools Sampler. *Prentice-Hall 1987*

- **O'Brien D. and Pitts D.** Korn Shell Programming by Example. *QUE 2001*

- **Quigley E.** Unix Shells by Examples (Csh, sh, ksh, awk, grep and sed). *Prentice-Hall 1999*

- **Rosenblatt Bill** Korn Shell. *O'Reilly & Associates 1993*

- **Wall L., Christiansen, T, Schwartz, R. L.** Programming Perl *O'Reilly & Associates 1996*

- **Welch B. B.** Practical Programming in Tcl and Tk. *Prentice-Hall 1997*

Index