Methods of Programming DV2

Scripting languages

Scripting

- Scripting is a way of automating a sequence of commands to a computer system. (Shell commands, utility program commands…)
- Both text and graphical interfaces can be scripted.
- In its simplest form a script is a list of commands.
- To make it possible to repeat commands and choose different commands in different situations, many scripting languages include programming language features (data and control structures).
- The basic design of a scripting language can be command or programming oriented.
- Many recent scripting languages are more oriented towards rapidly constructing simple programs rather than executing commands.

Why (not) use scripting

- Automate command sequences.
- Pre-process input to or post-process output from programs.
- ”Glue” different programs together.
- Easy and fast to write and run scripts for simple functions.

- Slow execution of the scripts themselves.
- Unsuitable for large programs or functions which require complex datastructures or algorithms.
- Dubious portability, language definition connected to a particular implementation and version.

- Use care when determining whether to use a scripting language or a normal programming language.

Typical features of scripting languages

(But certainly not every language has every feature!)

- Design motivated by the need to achieve something rather than sound design principles – no proper language definition.
- Very high-level in the problem domain (e.g. regular expressions).
- Poor and/or complicated data structures.
- Awkward syntax.
- Features with you have to try out to understand how they work.
- Interpreted languages – no compilation step but slow execution.
- Variables used by textually substituting their values into commands. (Error prone and a security risk!)
- Unsafe (no variable declarations, strong typing etc.)

Some scripting languages

sh	Unix shell. The classic! Ugly programming facilities built on top of the command function of the shell.
csh	Unix shell with C-like programming facilities.
tcl/tk	Programming language used to build GUIs. Extensible and embeddable.
perl	A sh/sed/awk superset with many features, awkward syntax and semantics.
python	High-level OO language. Good design. Inspired by functional and list processing languages

Useful programs in shell scripts

- `awk` Process text files
- `sed` ”Stream editor” of text files
- `fgrep, grep, egrep` Search files for regular expressions
- `wc` Count lines, words, characters in files
- `sort` Sort files
- `uniq` Remove duplicate lines in files
- `cat` Copies/concatenates files
- `tee` Split a pipe

Refer to the appropriate man pages for more info!

A simple shell script

Print pdf files on a printer

```
#!/bin/sh                (Identifies the file as a sh script)
for file in "$@"; do
   echo $file
   pdf2ps "$file" - | lpr -Ppr1332 -o raw
done
```

Sample usage:

```
lhe@harpo.it.uu.se> pdfpr *.pdf
fm.pdf
manual.pdf
lhe@harpo.it.uu.se>
```


How to run shell scripts

- The simple way: Run `/bin/sh` as a command and give the name of the script file as argument.
- The convenient way: To run a script as a Unix command (as in the previous slide), put the script file in a directory where the shell looks for commands (directories in the `PATH` environment variable).
- Script files need a special ”execute” permission to run. The ordinary read permission is not sufficient. Do the Unix command `chmod ugo+x` *`filename`* on your script files. (Or just `chmod u+x` if you don't want other people to run your scripts.)
- As there are different Unix shells with incompatible languages, you want to make sure that your script is run by `sh`. The first line in the sample script of the previous slides ensures this.

Features of the Unix shell language

Pipelines – glue commands together: `cmd1 | cmd2 …`

I/O redirection – change standard in/output: `cmd <file1 >file2`

Pattern matching file names: `cmd a? b*`

Background execution: `cmd &`

Comments: `#`

Command lists (instead of separate lines): `cmd1 ; cmd2 …`

Control structures: `if`, `while`, `for`, `case` …

Variable assignment: `var=value`

Variable substitution: `cmd $var`

Refer to the `sh` man page for details!

Variable handling

Variable assignment command: `var=value`
Variables also set by some shell commands.
Environment variables are imported into the shell script.

Variable contents are substituted into commands: `$var`

Some special variables:

`$0`	script name
`$n`	(a number >0) script argument
`$#`	number of script arguments
`$*` or `$@`	all script arguments (see next slide)
`$?`	return status of last command

Use `${var}` to prevent characters after the variable to be taken as part of the variable name, e.g. `${a}bc`.

Argument quoting

The arguments of a command are parsed *after* variable substitution. Spaces and other special characters (e.g. `*`) in the variable value will be treated as argument syntax!

Double quotes will prevent this.
Single quotes will even prevent variable substitution.

```
a=*
ls $a        …will list all files!
ls "$a"      …will list the file named "*"!
ls '$a'      …will list the file named "$a"!
```

Proper argument quoting is crucial!

`"$*"` …gives all script arguments within quotes.
`"$@"` …gives all script arguments *individually* quoted.

Control structures

`if` *`list`*`; then` *`list`*`; [else` *`list`*`;] fi`
The status (return) value of the last command in the list after `if` determines the test. (0 = true!!)
`case` *`argument`* `in` *`pattern`*`)` *`list`*`;;` etc. `esac`

The patterns work like file name patterns, except that the case argument is matched.

`while` *`list`*`; do` *`list`*`; done`

`for` *`var`* `in` *`arguments`* `do` *`list`*`; done`
The command list is executed for each argument in turn, with the variable var assigned that argument.

(Note: newline can be used instead of the single semicolons)

The test command

`test` *test-arguments* or special syntax

`[` *test-arguments* `]` (the spaces are mandatory!)

…carries out a test and returns zero (true) status if the test is true.

`-r` *filename*	the file exists and is readable.
s1 `=` *s2*	*s1* and *s2* are equal (the spaces are mandatory!)
s1 `!=` *s2*	*s1* and *s2* are not equal (spaces again…)
s1	*s1* is not empty.
n1 `-gt` *n2*	*n1* is a greater number than *n2*.
t1 `-a` *t2*	The tests *t1* and *t2* are both true.
t1 `-o` *t2*	Either of the tests *t1* and *t2* are true.
`!` *t*	The test *t* is not true.

…and many more! Do `man test`!

Other useful shell commands

`echo` *args*	write the arguments to standard output
`set -x`	write the following commands as they are executed
`shift`	delete the first script argument, shift down the others. ($2 becomes $1 etc…)
`read` *var*	read a line from standard input, set the variable.
`getopts …`	process option (–) arguments to the script.
`exit` [*n*]	stop the script (with status code `n`)

Again, see the man pages for more info.

grep

”Globally look for regular expression and print.”

`grep` *regexp filenames*

Searches the file(s) for lines where part of the line matches the regular expression and prints those lines.
Returns nonzero status if no lines are found so it can be used as a test. (Use `>/dev/null` to suppress output.)

A simpler (and faster) variant which searches for fixed strings:

`fgrep` *string filenames*

A variant with an extended form of regular expressions:

`egrep` *extregexp filenames*

Regular expressions

Regular expressions are patterns that match a string.
The most common forms of regular expressions in Unix utilities are:

`.`	match ny single character
`[a-cx]`	match one of a list (or range) of characters. Here a, b, c, x.
`[^…]`	as above, but matches a character *not* one of those given.
`^`	matches the beginning of a string
`$`	matches the end of a string
`…*`	matches zero or one occurences of the r.e. to the left
	other characters match themselves

Several regexps can be combined to form a sequence.
E.g. `^.[a-zA-z][a-zA-z]*` matches a string beginning with an arbitrary character followed by a nonempty sequence of letters.
Do `man 5 regexp` for more info.

awk

- `awk` reads a file (or standard input) one line at a time, processes the line and possibly writes to standard output (a read-process-print cycle). Every line is broken up into fields which can be accessed separately.
- An `awk` script has a number of commands of the form
  `pattern {action}`
- The action is performed for each line which matches the pattern. Without a pattern, each line matches, without an action the line is printed.
- Actions can set and use variables.
- Actions have a C-like syntax.
- `awk` is run using `awk` *script* or `awk -f` *scriptfile*

Some awk scripts

Printing lines longer than 72 characters
```
length > 72
```
Printing first two fields in opposite order
```
{ print $2, $1 }
```
Same, with input fields separated by comma and/or blanks and tabs
```
BEGIN { FS = ",[ \t]*|[ \t]+" }
      { print $2, $1 }
```
Adding up first column, print sum and average
```
      { s += $1 }
END   { print "sum is", s, " average is", s/NR }
```
Printing fields in reverse order
```
{ for (i = NF; i > 0; --i) print $i }
```
Printing all lines beginning with a start match until a stop match (repeating)
```
/start/, /stop/
```
Printing all lines whose first field is different from the previous one
```
$1 != prev { print; prev = $1 }
```
Printing a file, filling in page numbers
```
/^Page$/ { $2 = ++n; }
         { print }
```

sed

- `sed` (stream editor) applies editor commands to a file (or standard input) and writes to standard output. The commands basically affect one line at a time.
- An `sed` script has a number of commands of the form
  `[addr1[,addr2]] command [arguments]`
- The commands are performed in sequence for each line.
- Commands may be restricted to apply to one line or a range of lines identified by line number or pattern matching.
- `sed` is run using `sed -e` *`script`* or `sed -f` *`scriptfile`*.

See the man pages for more info. (Heard that before?)

An output file needs to be cleaned up…

```
HeerHugo. version dd. 10/6/96
CHECK CONSISTENTIE:
There are now 3 active variables
There are now 2 triples in use
...
Satisfiable
Listing current assignment
a == 1___
b != 1___
Total time used:      0.01
```

We want something more suited as input to another program:

```
a
~b
```

And in a separate file

```
time(0.01)
```


A sed script to clean up the output

```
/Total time used:/{ s/Total time used: */time(/
                    s/$/)/
                    w Atime
                    d
                   }
1,/Listing/d
s/== 1___//
/!= 1___/{ s/!= 1___//
           s/^/~/
         }
```

`{…}` groups commands
`s` is a string replacement command
`w` write the current line to a file
`d` deletes the current line.

A more complicated shell script

Replace a string in a file (set of files). The script file is named `subst`

```
#!/bin/sh
from="$1"
to="$2"
shift 2
for file in "$@"; do
  echo "$file"
  sed -e "s^$from^$to^g" "$file" > "$file.tmp"
  rm "$file"
  mv "$file.tmp" "$file"
done
```

```
$ more bar
kalle anka
$ subst kalle arne bar
bar
$ more bar
arne anka
```